AD-A120 056 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
REAL TIME STATUS MONITORING FOR DISTRIBUTED SYSTEMS.(U)
AUG 82 Z SEGALL, A SINGH, R SNODGRASS DASG60-80-C-0057

UNCLASSIFIED NL

1 OF 2

AD A
120056

Real Time Status Monitoring For Distributed Systems

Z. Segall, D. Siewiorek,
A. Singh, R. Snodgrass

27 August 1982

FINAL REPORT

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

Real Time Status Monitoring For Distributed Systems

Z. Segall, D. Siewiorek,
A. Singh, R. Snodgrass

27 August 1982

FINAL REPORT

MULTIPROCESSOR PERFORMANCE EVALUATION GROUP

Contract DASG-60-80-C-0057

Supported by the Ballistic Missile Defense Advanced Technological Center under contract ~~DASG60-81-007~~7. The views and conclusions contained in this ducument are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the BMDATC or the U.S. Government.

Table of Contents

List of Figures

1. Research Review

This is the Final Report of research done at Carnegie-Mellon University by the Performance Evaluation Group for the Balistic Missile Defense Advanced Technological Center, under contract DASG60-81-0077.

The area of research is instrumentation for distributed computing systems, and the period of research is January 1981-August 1982.

In order to facilitate the transfer of technology between Carnegie-Mellon University and Balistic Missile Defense Advanced Technological Center, three types of documents are provided. First, we provide status reports on Status Monitor, Synthetic Workload Generator (Pegasus) and Dormancy Study. These are updated versions of the status reports presented in the Interim Report. Second, the backbone of the report consists of the following three papers accepted for publication or presentation by IEEE Transaction on Computers, THe 3rd Conference on Distributed Computing Systems and Computer Magazine, respectively:

- An Integrated Instrumentation Environment for Multiprocessors
- Synthetic Workload Generator for Experimentation with Multiprocessors
- *Cm** - testbed

The exposing of the instrumentation concepts to the refereeing process is an integral part of our concern for assuring the quality of research done under this contract for BMDATC.

Third, we provide a collection of documents concerning the specification, design and implementation of the first stage Status Monitor. These documents present a detailed snapshot of the design decisions and implementation problems faced by us in solving the problem of Monitoring Distributed Systems.

1.1. Status of the Monitor

Work on the monitor has concentrated on three aspects: furthering the conceptual design, implementing the lower level mechanisms of the monitor and designing and implementing the relational monitor. At this point, we have a fairly complete idea of the tasks the various components perform and how these components will interact. The components are:

StarMon	low level data collection under the StarOS operating system on CM*, consisting of two processes:
	Accountant — interfaces to the Simon Accountant via the EtherNet;
	MonProc — performs name translation, enabling of events and miscellaneous services.
Medic	low level data collection under the Medusa operating system on Cm*;
Simon Accountant	interfaces with the resident monitor (either StarMon or Medic using a system-independent protocal;
Simon	the 'computing engine' for deriving high level information from event records;
Control	accepts queries from the user in a declarative language and translates these queries into update networks for Simon.

At this point, the first three components are nearing completion. Once their condition is stable, sensors will be placed throughout both StarOS and Medusa to provide a source of event records for Simon. The structure of Simon has been implemented, although more work is necessary. The Control component has been partially designed and is in the early stages of implementation. Also the a Sensor Definition facility has been designed and implemented.

Although this structure has been anticipated since early in the project, there were several surprises during the implementation stage. In particular, we found that

- specifying several parameters for a large number of sensors becomes a software engineering problem requiring automatic assistance;
- ensuring that the ethernet protocol was system-independent, yet efficient, was rather difficult;
- the straightforward implementation of the update network within Simon was several orders of magnitude too slow.

These issues will be dealt with in more detail in the remainder of this chapter.

1.1.1. Software Engineering Aspects of Sensor Specification

During the initial development of the low level event collection mechanism for the Cm* monitoring system, it became apparent that there were several procedural difficulties in the placement of sensors in the StarOS and Medusa operating systems. One difficulty was that the Sensor macro (which generates the event records) was becoming quite cumbersome. One design required twelve parameters for a typical sensor involving three domains! Since the sensors were to be placed in critical portions of the operating system, there was little room for error in the specification of these parameters. A second problem was the assignment of event numbers; an incorrect event number in a Sensor macro would result in the absence of event records of that type--a situation that might be difficult to detect by the user interacting with Simon. Two sensors with the same event number would cause havoc within Simon. A third problem is maintaining consistency between Simon's view of the world and the world as it actually is. This is especially tr during the early development of the monitor, when the collection of sensors inside the operating syste and the various attributes of those sensors, is changing frequently.

Finally, all these problems are exacerbated by the sheer number of sensors: we can easily envis several *hundred* sensors in both StarOS and Medusa when the monitoring system has been fully developed. The task of ensuring that all of these sensors, which are distributed among many source files, are correct and consistent, both within each other and with Simon, is unmanageable if it remains a manual one.

The solution was to create a database, called the *sensor description file (SDF)*, which contains information on the sensors defined in a given taskforce. As work progressed on designing the sensor description file, it became apparent that the syntax (as well as the programs processing that syntax) were general enough so that databases could be designed to contain information other than that related to the sensors. In fact, the *description files* will be used to communicate *all* static information to Simon from other programming environment tools.

1.1.2. A System-Independent, Efficient Protocol

The basic thesis of this research into monitoring is that the monitor must be capable of accepting high-level, non-procedural queries of system behavior and transforming these queries into an executable format, called an update network, using a wide variety of knowledge concerning the monitoring task. Although some of this knowledge must be of a system-dependent nature, it is vital that the algorithms and data structures which are using this knowledge be system independent. If the effort that goes into building a knowledgeable monitor must be repeated for each new system to be monitored, this effort will not be cost-effective.

There are several areas where this goal of system-independence was hard to achieve:

- naming;
- assumptions within Simon concerning the operating system;
- assumptions within Simon concerning the data collection mechanism.

In each case, the solution involved imbedding an abstract model of the environment in Simon, with the resident monitor (either StarMon or Medic) performing the translation to the operations supported by the operating system. As an example, consider naming, where the problem is in producing system-independent names for operating system objects. The model used by Simon assumes names are uninterpreted 21-bit integers which can be passed over the EtherNet to or from the resident monitor. StarMon uses a 16-bit field in the name to extract a capability for the object from an internal capability list. Medic, on the other hand, can use the nam;e to directly arrive at the address of the object. The main consideration is that the abstract model used by Simon must allow such a translation.

1.1.3. Efficiency of the Update Network

The first implementation of the update network was straightforward and, unfortunately, several orders of magnitude too slow. The metric is event records per second, with the first implementation executing about 2 event records per second. With each computer module in Cm* generating 2-5 event recods per second (a minimally acceptable rate), Simn must contend with 80-250 event records per second from all of Cm*. After several weeks of work, a rate of around 25 event records per second was achieved, still an order of magnitude too slow. At that point, several other systems having similar characteristics were considered, and a mechanism which could handle approximately 200 event records per second was designed using the results of a recently published thesis.

1.2. Synthetic Workload Generator--Progress Report

The conceptual design of the Synthetic Load Generator has been finalized. From an implementation standpoint, most of the Load Generator Environment has been implemented on both the Medusa and StarOS virtual machines (i.e. Cm* plus operating systems). A set of experiments have confirmed the initial feasibility of the tool.

An extension to the Synthetic Workload Generator enabling the tool to be used in the feasibility domain has been designed. The extensions are a software voting system and a fault inserter.

The following is the list of subtasks that are milestones for the project:

1. Functional Specification
2. Conceptual Design. This includes:
    - The design of the B language, a behavioral description language for the specification of the Load.
    - The design of the support Environment on the virtual machine for the B language.

   Design and implementation of the Load Generator on the Medusa and StarOS operating systems:

    - Design and implementation of the logical modules that form the Load Generator. This includes the modular decomposition of design and the specification of the inter-module relationships.
    - Design and implementation of the B language translator that maps the user specification to the virtual machine.
3. Experimentation with the Synthetic Load Generator
4. Modifications and extensions to the current design based on feedback from the experimental process, and the need to add additional functionality.

The first two milestones have been completed. The conceptual design of the Load Generator, as described in this report, is portable over message-based, object-oriented systems. Many of the concepts can be transported to other types of multi-computer architectures, including local area networks.

The design and implementation of the Load Generator Environment on both the Medusa and the StarOS operating systems has been essentially completed. Special care has been taken in the design to ease the translation process for the B language. Work needs to be done on designing the translator from the B language to the virtual machine. Like most Cm* support software, the translator will run on a PDP-10. At present, expertise exists to hand translate programs onto both operating systems. This has enabled us to make progress on the fourth milestone of the project.

Some experiments performed with the Load Generator have been discussed in this report. The primary objective is to demonstrate the usability of the Load Generator. The first experiment discusses a real-time application similar to the BMD computation. Some measurements on monitored queue are presented. The second experiment attempts to characterize the communication throughput of the Medusa operating system for varying parameters of the experiment. The initial experience confirms the feasibility of the Synthetic Load Generator.

1.3. Dormancy Study

Introduction

A hard failure study of over 276×10^9 dormancy part hours (11×10^9 on integrated circuits) and 118×10^9 power cycles (30×10^9 on integrated circuits) was conducted.* Based upon the data, a model for dormancy and the effect of power cycles was proposed. Even though the data is old (i.e., prior to 1972) and power cycling effects are difficult to quantify (i.e., how do you attribute a failure to power-turn-on stress as opposed to normal power-on stress), the conclusions that can be drawn from the data provide interesting insights.

The total System Life (SL) failure rate is modeled by four contributing factors

$$\lambda_{SL} = r_S \lambda_S + r_D \lambda_D + N_C \lambda_C + r_E \lambda_E$$

where r_S, r_D, and r_E represent the fraction of time the system or component is in (power-off) storage, dormancy, and fully energized (power-on), respectively. Dormancy is defined as power levels less than 10% of fully energized. N_C is the number of power-on-off cycles per hour. λ_S, λ_D, λ_C, λ_E are the failure rates due to storage, dormancy, cycling, and energized, respectively.

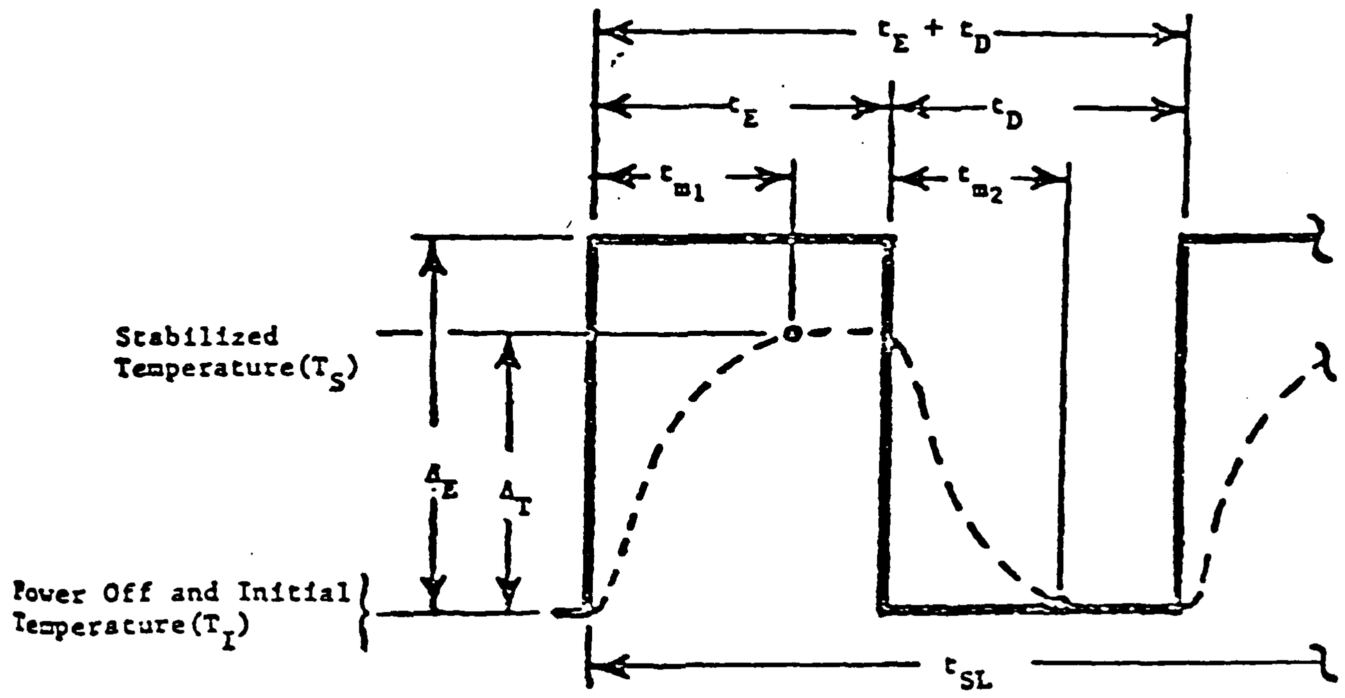
λ_S and λ_D are functions of screening class, environment, and complexity. λ_E is a function of screening class, environment, complexity, and junction temperature. λ_C is a function of screening class, environment, transient suppression, and temperature effect (C_T). C_T, as depicted in Figure 1, is a function of initial temperature, power derating, thermal lag, stabilized temperature, and residual temperature.

Summary

Table I summarizes the range of values for λ_D, λ_C, and λ_E as a function of screening class. Note that the rates are given in failures per 10^9 hours. The ratios of λ_E/λ_D and λ_C/λ_D are also given. The following conclusions can be drawn:

- According to the study, dormancy and storage failure rates are almost identical.
- One hour fully energized is equivalent to 10-50 hours of dormancy.
- The data for λ_C is less consistent than the other data. A rough estimate is that one power-on-off cycle ranges from negligible to equivalent to 30 hours of energized time. The most likely value is one cycle for 1-4 hours of energized time.

*J. Bauer, D. F. Cottrell, T. R. Gagnier, E. W. Kimball, "Dormancy and Power On-Off Cycling Effects on Electronic Equipment and Part Reliability," RADC-TR-73-248, Rome Air Development Center, Griffiss Air Force Base, New York, August, 1973.


$t_E + t_D$
t_E
t_D
t_{m_1}
t_{m_2}
Stabilized Temperature(T_S)
Δ_E
Δ_T
Power Off and Initial Temperature(T_I)
t_{SL}

NOTES:

——Energy Profile for Power On–Off Cycle

- - -Temperature Profile for Power On–Off Cycle

Δ_E = Energy Change (Power Off to Power On or Vice Versa)

$\Delta_T = T_S - T_I$ = Maximum Temperature Change

If $t_{m_1} \geq t_E$, then full Δ_T is not realized and this reduces temperature effect.

If $t_{m_2} \geq t_D$, then residual temperature effects increase temperature effect.

Figure [illegible]-1 General Diagram of Contributors to the Temperature Effect Factor C_T During Power On–Off Cycling

Table I. Summary of Failure Rates and Failure Rate Ratios

Screening Class	λ_D Dormant Failure Rate x 10^{-9}	λ_C Cyclic Failure Rate x 10^{-9}	λ_E Energized Failure Rate x 10^{-9}	$K_{C/D}$ (λ_C/λ_D)	$K_{E/D}$ (λ_E/λ_D)	$K_{C/E}$ $(\lambda_C/\lambda_E$
Class A	.17-1.33	.45	1.5-15	30-250	6-11	3-30
Class B	2-2.6	.03	40	.01	15-25	.001
Class C	3.08-5	15	80	3-5	15-25	5
Mil Std	4.7	-	166	-	35	-
Non Mil Std	10-15	1000	500-700	60-100	50-70	1.5-2

To see the impact of power-on-off cycling on expected equipment life, let $r_S = r_D = 0$, $r_E = 1$. Then

$$\lambda_{SL} = N_C \lambda_C + \lambda_E$$

or

$$\frac{\lambda_{SL}}{\lambda_E} = N_C \frac{\lambda_C}{\lambda_E} + 1$$

Hence, the MTTF compared to no power cycling is:

$$\frac{1}{N_C \lambda_C/\lambda_E + 1}$$

Conclusion

Table II illustrates the fraction of MTTF achievable as a function of N_C and $K_{C/E}$. Note that for one cycle per hour (such as the use of an LSI-11 in student laboratories), MTTF is significantly decreased almost inde endent of the value of $K_{C/E}$. For one cycle in 100 hours (such as a minicomputer turned on for a business week), MTTF is not significantly affected, regardless of the value of $K_{C/E}$. MTTF is a strong function of $K_{C/E}$ in the range of one power cycle per day and further data would be required to determine whether power cycling is a problem.

Table II. Fraction of MTTF Realizable as a Function of Cycle Rate and $K_{C/E}$

	λ_C/λ_E		
K_C	2	5	30
1	.333	.167	.034
0.1	.834	.667	.25
0.01	.98	.95	.77

2. An Integrated Instrumentation Environment for Multiprocessors

Zary Segall, Ajay Singh, Richard Snodgrass,
Anita K. Jones, Daniel P. Siewiorek

Abstract

This paper introduces the concept of an Integrated Instrumentation Environment (IIE) for multiprocessors. The primary objective of such an environment is to assist the user in the process of experimentation. The emphasis in an IIE is on experiment management (including stimulus generation, monitoring, data collection and analysis), rather than on techniques for program development as in conventional programming environments. We believe the functionality of the two environments should eventually be provided in one comprehensive environment.

An experiment *schema* is introduced as an appropriate structuring concept for experiment management purposes. *Schema instances* capture the results of an experiment for later analysis. An example is developed in some detail to demonstrate the potential benefits of such an approach. The three primary components of the IIE, namely, the *Schema Manager*, the *Stimulus Generator*, and the *Monitor*, are briefly described. A preliminary implementation of the design on the Cm* multiprocessor is briefly discussed.

Key words and phrases: *experimentation, experiment management, instrumentation, monitoring, multiprocessor performance evaluation, programming environment, stimulus generation, workload generation, automated testing.*

2.1. Introduction

Multiprocessor designs have long been proposed to meet the need for powerful, cost-effective computers. Several multiprocessors have been built to study the various trade-offs inherent in this approach [30, 73, 20, 58, 81, 65, 2, 68, 21]. An important objective of experimentation in performance evaluation and reliability is to provide evidence to validate the design decisions of these systems. Due to the increased number of independent components in multiprocessors, the space of possible experiments for such machines is orders of magnitude larger than for conventional uniprocessors. There is, therefore, a need to approach the problem of experimentation on multiprocessors in a structured manner.

Instrumentation of the machine is the first important step. Typical instruments discussed in the literature include software, hardware, hybrid, and computer network monitors, natural and synthetic workload generators, data compaction tools, and data analysis packages [22, 40, 8, 43, 71, 19, 47, 54, 36]. Most systems possess multiple instruments which have been built independently over a period of time with little effort toward integration. This unstructured approach has several disadvantages. First, an experimenter has to communicate with each instrument through its unique user interface, requiring familiarity with several sets of conflicting conventions in syntax and data formats. Second, data from one tool has to be converted manually to the format requirements of any subsequent tools. Furthermore, to make correlations across experiments, the experimenter has to manually keep track of experiment dates, input parameters, monitored results, system configuration, and so forth. Finally, getting the tools to interact during the course of the experiment is usually impossible.

Work in the direction of integrating instruments is found primarily in the area of computer network monitors [47]. Nutt [48] observes that the techniques for gathering measurement data have not been effectively used. Although the raw power of existing tools is quite adequate, the use of these tools is often so complex that experiments cannot fully utilize their functionality.

This paper recognizes the need for better human-engineered environments for experimentation with multiprocessors. It introduces the concept of an *Integrated Instrumentation Environment* (IIE) as a structured approach to facilitate the process of experimentation. The design presented emphasizes the integration of several instrumentation tools, including stimulus generation and monitoring, into a unified experiment management environment. An experiment script (a *schema*) is introduced as an appropriate structuring concept for experiment-management purposes. *Schema instances* are introduced to capture the results of an experiment for later analysis. A preliminary implementation of the design on the Cm* multiprocessor [30] under both the StarOS [34] and Medusa [51] operating systems is briefly discussed.

Although some program management concepts have been borrowed from conventional programming environments (PEs) [24, 77, 46], the thrust in the IIE is substantially different from what is typically discussed in conjunction with programming environments. The emphasis in an IIE is on experiment management, from stimulus generation to monitoring data collection and analysis, rather than on techniques for program development. We believe the functionality of the two should eventually be provided in one comprehensive environment. The IIE draws on the functions provided by PEs such as program specification and translation, version control, multiple programmer support, and module management. This paper assumes the existence of a PE and will therefore not discuss such functionality in the IIE.

Section 2.1.1 presents the functions to be performed by the IIE. The basic components of the design of the IIE are presented in Section 2.2. Stimulus specification and representation is discussed in Section 2.2.1. Section 2.2.2 discusses the techniques used to collect and process monitoring information. The run-time environment, presented in Sections 2.2.3 and 2.2.4, is a system-specific component permitting remote monitoring and stimulus control. Section 2.2.5 discusses the schema manager as the central control component supporting the execution of experiments. A preliminary implementation of the IIE on Cm* is discussed in Section 2.3. Relevant portions of an example are discussed throughout the paper to illustrate some of the concepts.

2.1.1. Functionality of the IIE

An Integrated Instrumentation Environment (IIE) consists of a set of tools which cooperate closely and present the user with a single uniform interface in order to assist and partially automate the process of experimentation. The general objective of an experiment is to inquire about performance, reliability, or any of a number of interesting properties of a computation. In the context of a computer system an experiment is the execution of an instrumented program in a controlled environment allowing measurement, collection, and analysis. An experiment may involve multiple executions of the instrumented program with different input parameters or within different environments.

The IIE supports the notion of an *experiment schema* as the high level unit of experimentation management. Each schema specifies a related collection of *runs*, that is, executions of an instrumented program. Intuitively, a schema can be seen as an parameterized experiment script, describing the experimentation process. A schema specifies the instrumented program, the monitoring directives, the specifications of the run-time environment, and the input parameters for each run.

The result of an execution of a schema is captured in a *schema instance*, containing measurements, values of schema parameters and environmental information. This is a data structure representing the unit of management for the experiment results. Schema instances are archived in a database for later analysis.

By using the generic notions of schema and schema instance the experimentation process can be expressed as in Figure 1.

```
Schema = DESIGN(Experiment)
WHILE (Not End of Experiment) DO
 BEGIN
   EXECUTE(Schema)
   CREATE(Schema Instances)
 END
ANALYZE(Schema Instances)
```

Figure 1: Experimentation Process in the IIE

Each phase of the experimentation process will be discussed in detail in the following sections.

An IIE requires software to support the several phases of experimentation, including

- Translation of collections of user-defined modules and predefined synthetic actions into instrumented parallel programs;
- Creation of the schema by merging the instrumented parallel stimulus, the monitoring directives and the environment information;
- Schema interpretation and run-time control;
- Creation of schema instances; and
- Analysis of schema instances.

In order to illustrate further the experimentation process described above, we will follow an example through in some detail. This example shows how the IIE, at each stage, interacts with the user, performs the required actions and generates its outputs. each stage. The example stimulus, called, simply, "a multiprocessor experiment", or MPX, involves a single *initiator* and multiple *servers* communicating through a shared buffer or mailbox. The initiator repeatedly sends requests through the buffer to one or more servers, which operate on those requests concurrently. When the buffer is empty, the servers wait for further requests; when the buffer is full, the initiator waits for a request to be removed by a server.

The servers perform identical functions, so a request can be satisfied by any server. Additionally, the servers communicate with each other via shared memory. The goals of the proposed experiment are to investigate

- the interaction between the request rate (expressed as the average number of requests per unit time) and the number of servers, and
- the effect of the requet rate and the number of servers on the average buffer queue length and the average waiting time in the buffer.

There are two interesting steady state behaviors that have different average queue length and service rate. In the first case, the request rate exceeds the aggregate processing rate of the servers, and hence, the buffer will always be full. In the other case, the buffer will always contain at most one request. The aggregate service rate will be approximately constant, yet radically different, in both cases. This analysis assumes a constant individual service rate by independent servers. However, in Cm*, accessing shared data perturbs the performance of both the servers and the buffer insert/remove operations in nonobvious ways, greatly complicating analytical modeling at the queue length and waiting time. As was shown above, the boundary between the two cases is quite distinct if contention is ignored. The experiment will investigate the boundary in the presence of contention.

To summarize our approach, experiments are described as schemata, and the result of executing a schema is a schema instance. The primary functions of the IIE are the creation of schemata, and schema management execution and control of schemata, along with the creation, management and analysis of schema instances. The next section presents the design of an IIE supporting these functions.

2.2. Design of the IIE

The IIE contains several components: a schema manager, a run-time environment, an instrumented stimulus and operating system, a database, and a monitor (see Figure 2). The monitor consists of a resident monitor, which gathers the data from the system under test, and a relational monitor, which aggregates and correlates the data into a high-level form. The user interacts directly with the schema manager, which communicates with the run-time environment and the monitor, which in turn interacts with the instrumented program (the *stimulus*) and the database. The IIE interacts with the PE through the database.

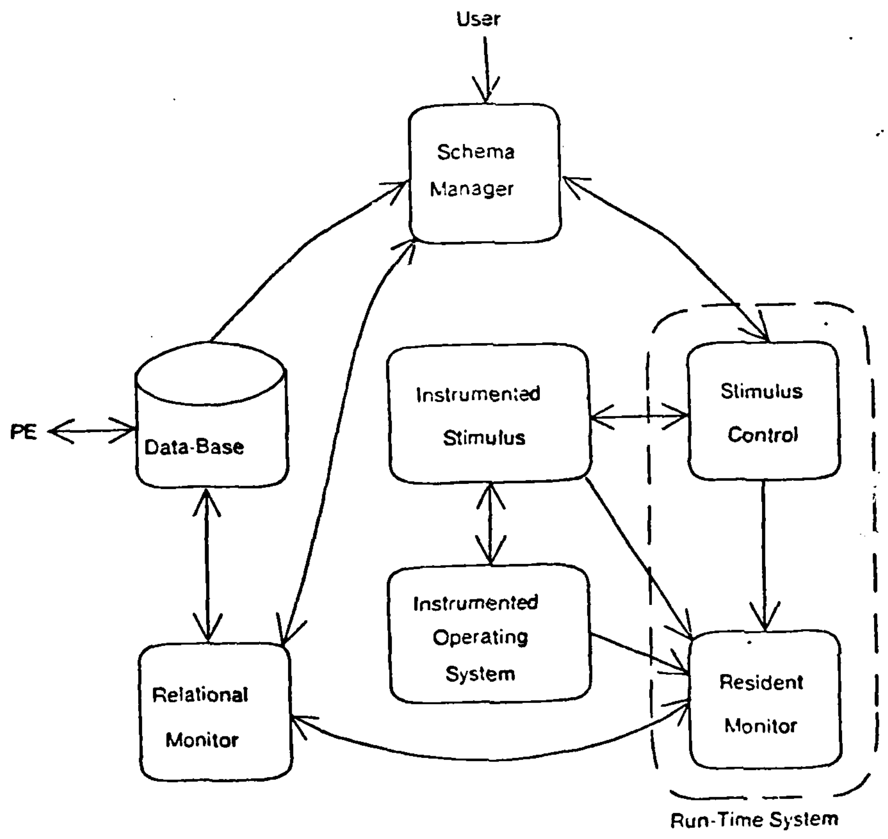


Figure 2: IIE Components
The arcs indicate transfer of data or control.

The schema manager is responsible for supporting the schema and schema-instance abstractions. The monitor initializes the schema instance with information specifying this environment, including details on the hardware configuration, the version of the operating system, support software, and stimulus, and the values of the parameters to remain constant for this execution of the schema. The schema manager then cycles through the runs as indicated in the schema, initializing parameters that vary on a per-run basis, starting the stimulus, and collecting the monitoring data. Finally, data concerning the runs as a whole is collected or computed, and stored in the schema instance for later study. Note that not all the IIE components should necessarily reside and execute in the same machine. In fact the Cm* IIE implementation spans several computer systems. The run-time system and the stimulus are resident in Cm*, whereas the schema manager, the database and the relational monitor are remotelly located in a VAX 11/780. The two computer systems are connected by an Ethernet link.

One motivation for partitioning the components of the IIE into a run-time environment and a remote environment is that only the run-time environment is constrained to any particular hardware or software configuration. Care has been taken to make the remote components as system independent as possible. Currently two preliminary implementations exist for the run-time environment for two different operating systems, while only one implementation of the remote components was necessary (see Section 2.3).

The stimulus controller component provides a well-defined interface to the instrumented stimulus. The functions it supports include modifying parameters within the stimulus, both before and during the run, generating initial control events for the stimulus, reporting errors back to the schema manager, and controlling the clock. Similarly the resident monitor provides a uniform interface for the relational monitor. The resident monitor is responsible for enabling and disabling *sensors* and for sending the information back to the relational monitor in a format convenient for further processing. The sensors are embedded in the stimulus, in the stimulus controller, in the operating system, and in the resident monitor itself. The relational monitor controls the resident monitor and computes derived information which is then stored in a schema instance in the database.

The database serves an important role in the IIE, because the information contained in the database is the end result of the entire experimentation process. Additionally, the interaction between the IIE and the PE occurs via the database by having one environment create objects in the database for the other environment to use. For instance, schemata are initially created in the PE, to be interpreted by the schema manager. Schema instances, created by the IIE, are managed using the version-control facilities of the PE. By using a common database, it is possible to make use of the functionality provided by the PE. This approach allows the designers of an IIE to concentrate on those operations unique to experiment management.

2.2.1. The Instrumented Stimulus: Representation and Specification

The stimulus is an arbitrary set of processes executing in parallel. The stimulus itself may incorporate sensors; in addition, sensors reside in the operating system and in the resident monitor. We have developed tools to aid in the rapid development of a stimulus. One of them is a workload generator. A user specifies the behavior of his parallel program in a special high-level behavior-description language, the *B-language*. This behavior is specified as a directed data flow graph, similar to a complex bigraph [10, 23]. The nodes of the graph represent subtasks, or processes, that execute in parallel with other subtasks. Each subtask is composed of *actions*, parametrized program fragments that may be predefined or user-defined, repeated at certain rates. Associated with each arc is a buffer which may hold data variables or control tokens flowing from one subtask to another. Each subtask has an associated control tuple (*i*, *o*), where *i* corresponds to the in-firing rule for the subtask and *o* corresponds to the out-firing rule. This set of firing rules characterizes the precedence relationship between the subtasks of the graph. A B-language program is compiled into an executable version as illustrated in Figure 3. This section gives a brief overview of the B-language; a more detailed discussion can be found in [66].

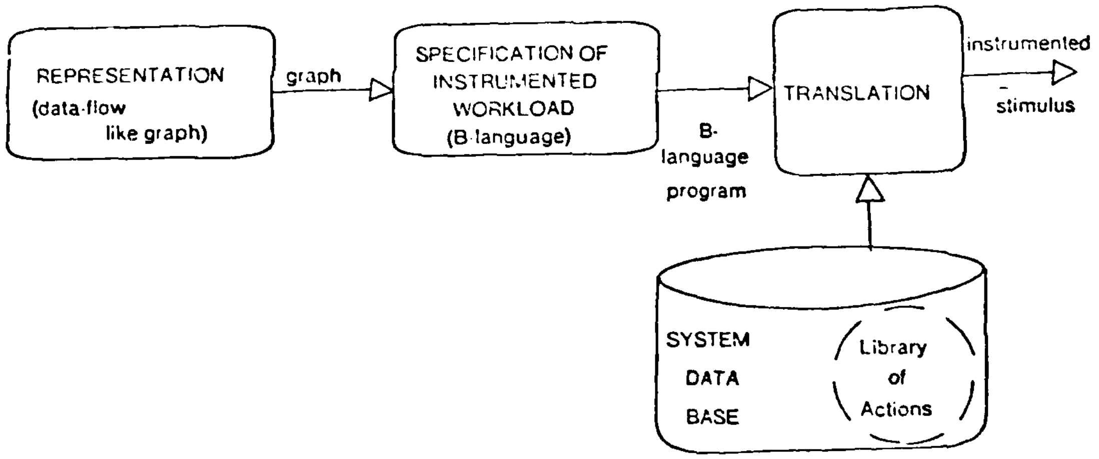


Figure 3: Stimulus Generation Steps

The B-language thus represents the interaction of parallel processes via the graph model of computation. A typical example is shown in Figure 4. Subtask *A1* is fired by the arrival of a token in buffer *B1* which corresponds to the entry arc of the graph. Upon completion, subtask *A1* fires either of subtasks *A2* or *A3* by placing control tokens in either buffers *B2* or *B3* respectively. There is a certain probability associated with the OR-output logic of subtask *A1* (designated by the '+'). Finally subtask *A4* fires if it receives a token either from *A2* or *A3*. Upon completion, it places a token in buffer *B6* which corresponds to the exit arc of the graph and represents the end of a single execution of the parallel synthetic program.

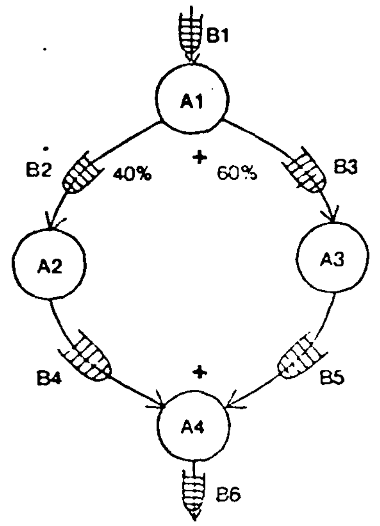


Figure 4: A Parallel Synthetic Program— Graph Representation

The B-language subtask declarations for this example are—

```
SUBTASK A1 {  INLOGIC : B1 ; OUTLOGIC : %40(B2) OR %60(B3) }

SUBTASK A2 {  INLOGIC : B2 ; OUTLOGIC : B4 }

SUBTASK A3 {  INLOGIC : B3 ; OUTLOGIC : B5 }

SUBTASK A4 {  INLOGIC : B4 OR B5 ; OUTLOGIC : B6 }
```

Notice that the buffers B_1 to B_6 correspond to the arcs of the parallel synthetic program. The delimiter "%" is used to specify the branching probabilities for the arcs of an OR-output.

The specification of parallel synthetic programs in the B-language is based on the object model supported by both operating systems on Cm* [34, 51]. The objects represented directly in the B-language include—

- The *task force* object: The task force abstraction, a collection of processes that cooperate to achieve a single logical task, is represented by a set of *subtasks*.
- The *subtask* object: This is the sequential computation unit that cooperates with similar user-defined objects to compute the overall stipulated multiprocess task.

- The *buffer* object: The buffer object is a conventional queue of messages and is used by the subtasks to communicate with each other.

- The *semaphore* object: Semaphores synchronize requests for shared resources.

- The *file* object: Files represent a sequence of bytes.

- The *shared* data object: Variables specified in the shared data object are globally shared by all the subtasks of the task force. This allows communication of data and control through shared memory.

- The *table* object: Tables implement functions varying with time.

Within a subtask, the basic building block is an action. To capture the cyclic nature of synthetic workloads, an action a_i itself is described by an action-repetition tuple (specified as $\langle a_i, r_i \rangle$). This tuple specifies that the action a_i is repeated sequentially r_i times, constituting action a_i. An action may be arbitrarily complex, and may be furthèr composed of action-repetition tuples. Also, both the a, and the r can be parametrized. Other control constructs within a subtask include composition and conditional and probabilistic branching.

The library of actions consists of a collection of predefined and user-defined program fragments, programmed in the systems programming language and stored as part of the system database. Examples of predefined actions include sending or receiving messages via a buffer, inputing or outputing to a file, referencing local memory, blocking on a semaphore, and accessing a shared resource. The user gains flexibility by being able to include his own special program fragment among the actions in the library. An example of a user-programmed action is the code for a disk process in a database application running on a specific multiprocessor. Hence, the library of actions is specific for a particular multiprocessor system. The B-language should be viewed as a portable framework into which system specific actions are inserted from a library of actions.

Special control constructs are included in the B-language so that the schema manager may control the user's workload at run-time as specified in the schema. The control commands initiated by the schema manager are executed by the stimulus controller component of the run-time system. The VARY construct in the language permits the stimulus controller to vary parameters on a per-run basis. The language also allows one to specify that the parameters are to vary in real time. This is accomplished by binding a real-time function to a run-time variable on a per-run basis. The real-time function is defined by a table object and an associated interval of time. The stimulus controller forces the run-time variable to take on successive values from the table during successive time intervals.

Using the MSGEVENT construct, the language permits the stimulus controller to initiate variable time-driven events in the stimulus on a per-run basis. This construct requests the stimulus controller to deliver messages to a buffer with inter-message time periods as specified by successive entries of a table. The stimulus controller can associate a different table object, or a constant time-period, with the MSGEVENT variable on a per-run basis.

To allow measurement of the generated workload, a special SENSOR construct permits a user to embed sensors into his program. Sensors allow specified information as well as a timestamp to be sent to the monitor as event records. In addition to user-defined sensors, the B-language program has some built-in sensors. For example, the start time and end time for each execution of a subtask are automatically recorded in the event record. Furthermore, instrumentation available in the operating system and the IIE run-time system allows the schema manager to access information not explicitly specified in the B-language program. An example is information regarding the interaction of the stimulus and the operating system.

The B-language translator constructs special data structures allowing the stimulus controller to exercise external control over the experiment as specified in the B-language program. The translator also generates sensor descriptions (see Section 2.2.4) for all programmed and predefined sensors in the B-language program. These descriptions are used by the relational monitor to sort out event records flowing from the resident monitor.

As an example consider the B-language program (Figure 5) for the single-requester, multiple-server experiment discussed in Section 2.2. The task force consists of an array of five identical server subtasks that wait on the RequestBuffer for queued service requests. The RequestBuffer is associated with the message-event generator via the MSGEVENT construct. This allows an experimenter to vary the request rate by changing the time (RequestPeriod) between successive firing of servers on a per-run basis. The BEGIN and END constructs mark the service loop of each subtask which is executed each time its in-firing rule is satisfied. In this example, each server does ten units' worth of work local to its processor, and then does some variable number of accesses to global data, which is arbitrated by a semaphore. A sensor, StartGlobalPhase, is embedded in each subtask and sends an event record to demarcate the transition from local work to global work. Built-in sensors record the begin and end of each subtask and the firing of request tokens by the message-event generator. The variable parameters of the experiment are the number of active servers, the request rate, and the amount of global work done by each server. This program will be specified in the schema as the stimulus for the MPX.

```
TASKFORCE MPXperiment ;
BUFFER
   RequestBuffer{ SIZE: 512 } ;
SEMAPHORE
   GDSemaphore{ INITIAL: 1 } ;
SHARED
   GlobalData[512] ;
VARY
   RequestPeriod;
MSGEVENT
   RequestService = RequestBuffer @ RequestPeriod ;
SENSOR
   StartGlobalPhase ;

SUBTASK Servers[1..5]
       { INLOGIC : RequestBuffer }
VARY
   SharedDataAccess ;
BEGIN
   <$DoLocalWork : 10>,
   StartGlobalPhase,
   <$AccessSharedData(GDSemaphore,GlobalData): SharedDataAccess>
END
```

Figure 5: B-language program for the MPX

2.2.2. Relational Monitor

In the IIE, each time the experiment schema is interpreted, and the stimulus executed one or more times, various monitoring information is collected and stored in the database in a schema instance.

The model of the monitoring data adopted in the IIE is a variant of the *relational model* used in conventional relational databases [79]. Information is recorded as a collection of two-dimensional tables, called *relations*. Each row, called a *tuple*, records a particular relationship between entities named in the columns, called *domains*, of the tuple. For example, the relation *Running (Process, Processor)*, with two domains, may contain the tuple *(MyProcess, ProcessorA)* indicating that the process called *MyProcess* is running on the processor called *ProcessorA*. Relations used in monitoring are temporal, in that each tuple records relationships that are true at an instance of time or over some interval of time. A relation involving instances of time is called an *event relation*; each tuple records the occurrence of a particular event. A *period relation*, on the other hand, records a relationship that exists for an interval of time. Periods are delimited by events; each tuple (period) in the *Running* relation is associated whith two other event tuples, one in the *Start* relation and one in the *Stop* relation. Time is included in an implicit domain manipulated by the monitor.

The *Running* relation is an example of a *primitive relation*, because the information contained in the relation is a direct translation of a set of recorded events. Primitive relations may be divided into three categories: operating system, stimulus control, and user-defined. The first category is concerned with information involving the operations and data structures supported by the operating system. The *Running* relation is in this category. The second category involves the actions performed by the run-time portion of the IIE. Examples of event relations from the MPX include

- *RequestService(TokenID)* : the sending of a MsgEvent token to the RequestBuffer; the TokenID identifies the token;
- *ServersStart(Index, TokenID)* : the in-firing of a sensor's subtask; the *Index* identifies the Server; the *TokenID* identifies the token causing the firing;
- *ServersEnd(Index, TokenID)*: the out-firing of a sensor's subtask.

The one user-defined primitive relation specified in the MPX, *StartGlobalPhase*, is also an event relation and contains only the implicit time domain. This relation was declared as a sensor in the B-language program for the MPX (see Figure 5), and records the time at which the Server subtask finished its local work and started the shared data access.

```
range of R is RequestService    ; references to R will indicate the
                                ; RequestService relation
range of S is StartServers
range of Sp is StopServers
define WaitingInQueue (R.TokenID)       ; one domain, the request's
                                        ; TokenID
        where R.TokenID = S.TokenID     ; the request is being serviced
                                        ; by a server
        start R                 ; the waiting begins when the request
        stop S                  ; is made, and ends when the server
                                ; starts
range of W is WaitingInQueue
define QLength (L = Count(W))   ; count the number of outstanding
                                ; requests in the buffer
range of Q is QLength
define AverageQLength (AvQL = Average(Q)) ; instantaneous average

define TotalWaiting (W.TokenID)
        where Sp.TokenID = W.TokenID
        start W                 ; total waiting time begins when the
        stop SP                 : request was made, and ends when the
                                ; server stops
range of TW is TotalWaiting
define ServiceRate (SRate = 1 / Average(Duration(TW)))
```

Figure 6: Queries for the MPX

Given a collection of primitive relations, new relations can be defined as a result of operations performed on existing relations. These *derived relations* are specified using a relational query languages.

The query language used in the IIE is a version of Quel [72] augmented with additional temporal constructs and is discussed elsewhere [69]. Figure 6 illustrates the definition of the derived relations `AverageQLength` and `ServiceRate` used in the MPX. The former relation has one domain, `AvQL`, with the tuples specifying this value for the various time intervals. Similarly, the `ServiceRate` relation will have one domain, `SRate`, containing values varying over time. These queries will be referred to by the schema for the MPX, and will specify both the primitive relations to be monitored and the calculations to be performed on the data in the event records.

2.2.3. Stimulus Controller

The stimulus controller component of the run-time system is a set of utilities that permit control of the stimulus as specified in the schema. While the schema manager provides experiment *management* through the management of the schema abstraction, the sti..ulus controller provides low-level experiment *control* through the management of a single run. The motivation was to separate the low-level control functions from the experiment-management functions so that different management strategies could be carried out using common control primitives. The functions exported by the stimulus controller are therefore geared towards the initialization and execution of a single run.

One responsibility of stimulus controller is to ensure the repeatable behavior of a run by eliminating side-effects from one run that might perturb the next run. An example of a side-effect is the presence of tokens left over in the edges (buffers) as a result of the previous run. The stimulus controller ensures that all data structures are in a well defined state at the beginning of a run. For example, buffers are emptied and all semaphores are be initialized as specified in the B-language program.

The stimulus controller is also responsible for the variation of parameters on a per-run basis and in real time during a run. The variation of parameters on a per-run basis involves the VARY parameters of the B-language program (see Section 2.2.1), and the variation of the graph-structure representation of the program. A typical modification of the graph structure involves changing the number of active subtasks for a particular run. This is particularly useful in real-time experimentation, where one wants to determine the number of subtasks necessary to meet real-time constraints. The variation of parameters in real time during a run involves the variation of the run-time variables of the B-language program according to some function of time expressed as a table object and an associated interval of time.

The stimulus controller must provide a well defined mechanism to start the run. In the graphical representation of the program this corresponds to firing the entry node, that is, placing a token on the entry arc of the graph. To start a run, the stimulus controller delivers a specified number of control tokens into a system-defined buffer, called the IgnitionBuffer, which corresponds to the entry arc of the data-flow graph. The user may now use this source of tokens to start any desired subtask, by specifying the IgnitionBuffer appropriately in the in-firing rule of that subtask. Similarly to detect the end of a run the stimulus component watches a system-defined TerminationBuffer for a specified number of tokens.

The stimulus controller has four major sub-components. The first sub-component executes basic control functions, including *initialize*, to initialize the instrumented program before each run; *fire*, to fire a specified number of tokens into the IgnitionBuffer; *vary*, to permit the variation of VARY-parameters on a per-run basis; *display*, to display the value of a *vary*-parameter; *enable/disable*, to enable or disable subtasks on a per-run basis; and *status*, to return the status of the program. Observe that functions such as *display* and *status* are interactive in nature and can be used during the interactive creation of a schema (see Section 2.2.5).

Second, a message-event generator delivers token messages to pre-specified buffers according to pre-specified functions of time. Control functions performed by this sub-component include *start generator*, to start the message-event generator for a particular run; *stop generator*; and *set message event*, to allow the association of either a table object or a constant with a buffer.

Third, a run-time variable driver ensures that all run-time variables vary in real time as specified by its associated table object and time interval. The main control function of this module is to allow the association of different table objects and time intervals with a run-time variable on a per-run basis.

Fourth, a clock module permits access to a set of clocks distributed over the system. This module is used by the message-event generator, the sensors, and the run-time variable driver.

Additional functionality in the instrumented program may be added by augmenting the stimulus controller. For example, a set of components used for experimentation related to reliability has been designed and partially implemented. This includes software-implemented voters, and accelerated fault-insertion and configuration-control modules.

2.2.4. The Resident Monitor

The monitoring information is collected as *event records*, generated by *sensors* in the instrumented stimulus, the run-time system, the operating system, or the hardware. Each event record contains an indication of the operation being monitored, the name of the component performing the operation, and the name of the object the operation is being performed on. The event record may optionally contain a timestamp and other information germane to the event. For instance, a sensor located in a file-system process might generate event records for file reads. In this case, the event record would include the name of this process, the name of the file being read, an indication that this is a file-read event, the timestamp, and perhaps the block number being read.

Highly selective filtering of the event records is necessary to constrain the flow of event records into the monitor. Enabling and filtering directives are encapsulated in data structures called *receptacles*, associated with either active components, such as a file-system process, or passive objects, such as a file. Receptacles contain event-enable switches as well as a buffer for temporarily storing event records. The resident monitor (and thus, indirectly, the relational monitor) has the ability to enable switches in each receptacle. The flexibility in associating receptacles with either processes or objects provides a mechanism for filtering the event records. For example, if the receptacle was associated with the file, and the file-read event was enabled, event records for all file reads performed on the file would be written into the receptacle. On the other hand, if the receptacle was associated with a file-system process, event records for all file reads performed by the process on any file would be written into the receptacle.

A task force is instrumented by specifying the sensors, events, and object types in a file called a *sensor description*. The operating system and stimulus controller, being task forces themselves, are also associated with sensor descriptions. A sensor description is generated automatically when a B-language program is processed. Users may also write their own sensor descriptions if they so desire. Figure 7 illustrates the sensor description generated from the B-language program for the MPX given in Figure 5. This description includes a sensor-process definition for each subtask and for the stimulus controller, and events for the start and end of execution of each subtask and the start of each run. Another program takes the sensor description and produces optimized code for each software-implemented sensor, based on the specifications in the sensor description. Sensor descriptions thus allow users to specify their own sensors which will utilize the same mechanisms for event record and generation as the sensors embedded in the run-time and operating systems.

It is important to note that the user never needs to be concerned about receptacles or event records. Instead, the IIE (through the monitor component) presents to the user the view of a database composed of temporal relations. New relations can be derived using the query language (identified in Section 2.2). As a result of executing a query, the appropriate operations (locating and enabling receptacles, processing event records, and generating the schema instances) are performed automatically.

The use of receptacles and sensors may extend from sensors implemented in hardware to sensors embedded in the operating system to sensors placed in the user's program. It is the resident monitor's

```
(Taskforce (name MPExperiment)                   ; Standard prelude
   ...  )
(SensorProcess (Name StimulusControl)
   ...  )
(Event (Name PerRun)
       (Domains (Domain (Name RunNumber)
                        (Type Integer))
                (Domain (Name RequestPeriod)
                        (Type Integer))
                (Domain (Name ServerCount)
                        (Type Integer)))
       (Timestamp yes)
   ...  )
(Event (Name RequestService)                     ; MsgEvents
       (Location StimulusControl)
       (Domains (Domain (Name TokenID)
                        (Type Integer)))
       (Timestamp yes)
   ...  )
(SensorProcess (Name Servers)                    ; SubTasks
   ...  )
(Event (Name ServersStart)
       (Location Servers)
       (Domains (Domain (Name Index)
                        (Type Integer))
                (Domain (Name TokenID)
                        (Type Integer)))
       (Timestamp yes)
   ...  )
(Event (Name ServersEnd)
   ...  )
(Event (Name StartGlobalPhase)                   ; User-defined sensors
   ...  )
...
```

Figure 7: Sensor Description for the MPX

responsibility to extract the event records from the receptacle and send them to the relational monitor. By the time the relational monitor receives the event records, they are in an identical format regardless of how they were generated.

2.2.5. Schema Management

The central management and control of the schema and the schema instances is performed by the schema manager. Functions of the schema manager fall into two broad categories: the creation, manipulation, and execution of the schema and the creation, archiving, and cross-analysis of schema instances. The schema manager is organized in three main functional parts:

1. A user interface provides a uniform view of the various components of the IIE. Schemata can be created using conventional text editors, or incrementally by directing the IIE to perform a series of runs. In the latter case, the corresponding schema and schema instance are

automatically generated and archived. This incremental mode is particularly helpful in the tuning of experiments. The user interface also directly supports monitoring queries and database queries thereby allowing a user to manipulate and analyze schema instances.

2. A schema interpreter scans the schema and sends control directives to the run-time system, including global initialization commands for the entire experiment along with commands to set up, start, and terminate each run.

3. A schema-instance generator interacts with the relational monitor to ensure that an instance is created and placed in the database. Both predefined and user-defined relations are created and stored in the schema instance as a result of interpreting the schema.

The schema contains all the necessary information to perform a complete experiment. It consists of five major components: the system configuration, the stimulus, monitoring directives, initial experiment conditions, and experiment directives (see Figure 8). The system configuration completely defines the environment the experiment is to be performed in. The stimulus is in the form of a translated B-language program containing controlling parameters and data-collection sensors as described in Section 2.1. The monitoring directives are in the form of a collection of queries as described in Section 2.2.2. The initial experiment conditions consist of a set of invocation parameters and the required resources (i.e. hardware and operating system configuration and instrumentation, stimulus version, etc.). Invocation parameters can be used to initialize parameter values for experiments and are typically specified at schema interpretation time. The experiment directives are interpreted by the schema manager and specify how the stimulus should be executed. Specifications are provided for the iteration of the stimulus over the experiment runs along with the variation of parameters for each run.

```
SCHEMA (<invocation parameters>)
       <system configuration>
       <stimulus>
       <monitoring directives>
       <initial conditions>
       <experiment directives>
END SCHEMA
```

Figure 8: High level organization of a schema

During schema execution, the relational monitor creates a schema instance to hold the results of the experiment, The monitor collects all the resulting event records together with the schema identification and environment information and creates an object to be managed by the PE. By using standard relational database queries, the user can then perform analyses across schema instances. The data in the instance which is collected automatically provides the user with enough information to replicate any particular execution of the schema to verify the results.

```
SCHEMA MPX (RequestPeriod, SDA)

SYSTEMCONFIGURATION <configuration data>;

TASKFORCE <B-language program>;

MONITORQUERIES <relational queries>;

RESULTRELATIONS AverageQLength, ServiceRate;

VARY SharedDataAccess[I] = SDA WHERE I FROM 1 TO 5;

VARY NoOfServers FROM 1 TO 5
DO
    BEGINEXPERIMENT
    ENABLE Server[I] WHERE I FROM 1 TO NoOfServers;
    TERMINATE AFTER 30 seconds
    ENDEXPERIMENT
OD
ENDSCHEMA
```

Figure 9: The Schema for the MPX.

In order to illustrate the use of schema and schema instance, consider the schema describing the MPX, shown in Figure 9. The schema has two invocation parameters, RequestPeriod and SDA. The configuration data specifies the resources requested by this experiment, including the versions of the operating system and IIE components, the hardware components, data files to be read by the stimulus, and initial tests to be used later to calibrate the results.

The experiment directives are in the form of a loop which generates the execution of 5 runs. Each run will have its own value for the `NoOfServers` parameter. The execution of this schema will terminate when 30 seconds have passed for each run. During execution, the sensors implanted in the B-language program will generate data which is collected according to the monitor queries.

Each time this schema is interpreted, a schema instance will be automatically created in the database by the IIE. Each instance will have the following components:

- The date, time, and user identification;
- The values of the invocation parameters;
- Exact version numbers of all software used in the experiment;
- A detailed description of the hardware configuration;
- Results of the initial tests as specified in the system configuration; and
- The system- and user-defined relations (in this case, the `PerRun`, `AverageQLength`, and `ServiceRate` relations).

Once the instances have been created, additional analysis can be performed on the instances

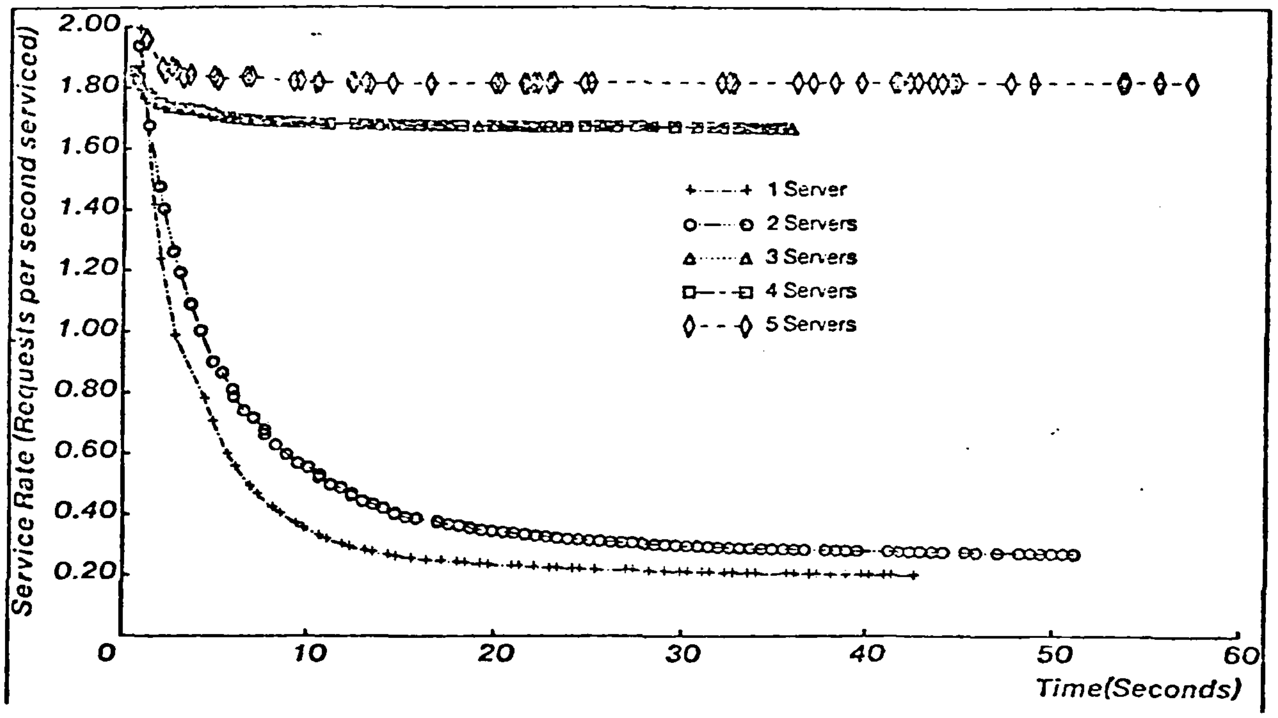


Figure 10: Service Rate vs. Time for a Variable Number of Servers

individually or as a group. Figure 10 shows the relationship between average service rate and time for a RequestPeriod of 200 milliseconds and a value of SharedDataAccess of 400 accesses per request. Initially the service rate is high, since the buffer is empty. For five servers, the buffer never contains many requests, so the average service rate remains high. However, for less than three servers, the buffer fills up quickly, causing the average service rate to plummet. The behavior with three or four servers is more involved, and further analysis is necessary using different values for the request period and the SDA.

2.3. Implementation

2.3.1. Background

Our research vehicle is the Cm* multiprocessor. Cm* is a 50 processor multiprocessor developed and implemented at Carnegie-Mellon University. Two operating systems, MEDUSA and STAROS, have been developed for Cm*. In addition, substantial utility software built for Cm* runs on other general-purpose computers.

2.3.2. Status

An initial version of the IIE has been partially implemented for Cm*. Two versions of the run-time system have been developed, one for each operating system [69, 66]. A substantial library of actions has accumulated for both operating systems, and work is proceeding on implementing the B-language translator. An initial version of the relational monitor has been developed, including the sensor-description of the processor, although substantial effort is still needed before general queries and multiple schema-instance analysis can be executed [69]. The schema manager is in the final design stages. It is expected that a full implementation of the IIE will be completed by the end of 1982.

2.4. Conclusion

The IIE constitutes a systematic approach to the task of experimentation on multiprocessors. This approach emphasizes the integration of the tools used for such experimentation and the development of techniques for experiment management. The tools incorporated into the initial design of the IIE have been oriented primarily toward performance measurement. Work is proceeding in the area of reliability experimentation, specifically to enhance the monitor so that it can function across system failures and to implement fault insertion into the stimulus in a controlled fashion. Another interesting use of the IIE is in automated testing of revised modules in the framework of version control. Future research areas include the integration of the IIE with a multiprocessor PE, the incorporation of hardware monitors and other tools into the IIE, and the development of an IIE supporting experimentation of real-time systems.

2.5. Acknowledgements

The authors would like to acknowledge the contributions of some of the concepts and of the implementation by the other members of the Multiprocessor Performance Evaluation Group: Xavier Castillo, Robert Chansler, Ivor Durham, Peter Highnam, Ed Gehringer, and Pradeep Sindhu.

3. Synthetic Workload Generation for Experimentation with Multiprocessors

Ajay Singh, Zary Segall

4. Abstract

Multiprocessors are relatively complex computer structures and are still undergoing an experimentation phase. An important aspect of this phase is experimental performance evaluation to understand and validate new and existing systems. An integral part of such experimental performance evaluation is the specification and generation of a controlled drive workload.

This paper presents the design of a controllable, interactive, synthetic workload generator for multiprocessors. The primary objective of such a tool is to assist in the process of experimental performance evaluation. This has been achieved in two ways. First, by designing a high level language for the representation of a parallel synthetic program as a data-flow graph. As part of this language, control of the workload is directly represented via special control constructs. Second, by supporting an experimentation environment on the multiprocessor which allows the user to control, vary, and measure his workload as specified in the language without having to recompile or re-debug his programs. A message communication throughput experiment is described to demonstrate the use of the workload generator.

Key words and phrases: *workload, performance evaluation, environment, experimentation, multiprocessors, parallel programs.*

4.1. Introduction

Multiprocessors have been long proposed to meet the need for rebust, powerfull cost effective computers. Being relatively complex structures, the space of design decisions is too large and complicated to be predicted and validate only analyticaly. There is, therefore, a critial need for experimental performance evaluation. An integral part of such experimental performance evaluation is the specification and generation of a controlled drive workload.

Here, by workload we mean [1] the collection of all individual programs and data that are processed by the computer system during a specified period of time. The term *workload characteristic* refers to demands placed on the system resources. Ferrari [19] classifies three basic techniques for generating a drive workload. First, *natural* workload generation, where the workload is generated by the real application. Second, *artificial* workload generation, where the generated workload is independent of the real application. And finally, *hybrid* workload generation, where the workload is generated by the manipulation of natural workloads. Many methods for generating workloads for uniprocessor systems are described in the literature [40], [35], [63], [82], [19], [71], [41]. Not much work has been reported on the problem of workload generation for multiprocessors [43]. This is primarily because of the lack of operational hardware and/or support software.

Owing to the difficulties in controlling and parameterizing natural workloads, artificial workloads are a preferred method for experimental performance evaluation. Examples of artificial workloads include, instruction mixes, kernel programs, benchmarks, and synthetic workloads. We feel that synthetic workloads have greater potential for experimental performance evaluation of multiprocessors. This is primarily because synthetic workloads are much more flexible and controllable since they include adjustable parameters to mimic a broad range of natural workloads. The problem of generating synthetic workloads for multiprocessors has two aspects:

- The *representativeness* of the synthetic workload, that is, how closely does the generated workload mimic the real workload.
- The *generation* of the synthetic workload, that is, what method should be adopted to specify and produce this representative workload.

This paper addresses the second aspect of the problem of synthetic workload generation for multiprocessors. It presents the design of Pegasus, a controllable, interactive, synthetic workload generator implemented on Cm*, a fifty processor multiprocessor at Carnegie-Mellon University. A more detailed discussion can be found in [67]. Our primary objective in designing and implementing a synthetic workload generator was to speed-up the experimental performance evaluation of Cm*. This has been achieved in two ways. First, by designing a high level language for the representation of a parallel synthetic program as a data-flow graph. Second, by supporting an experimentation environment on the multiprocessor which allows the user to control, vary, and measure his workload as specified in the language without having to recompile or re-debug his programs.

Recognizing the representativeness problem, and its difficulty in multiprocessors, we have limited our performance evaluation goals using parallel synthetic programs to the following:

- Parameterization of the multiprocessor system. Examples include quantifying speedup, throughput, response time, and resource utilization, as a function of the prameters of the system.
- Preliminary evaluation of an application on the virtual machine given its detailed decomposition and workload characteristics.
- Comparison of two different virtual machines along various dimensions, by comparing their behavior in the presence of similar workloads.

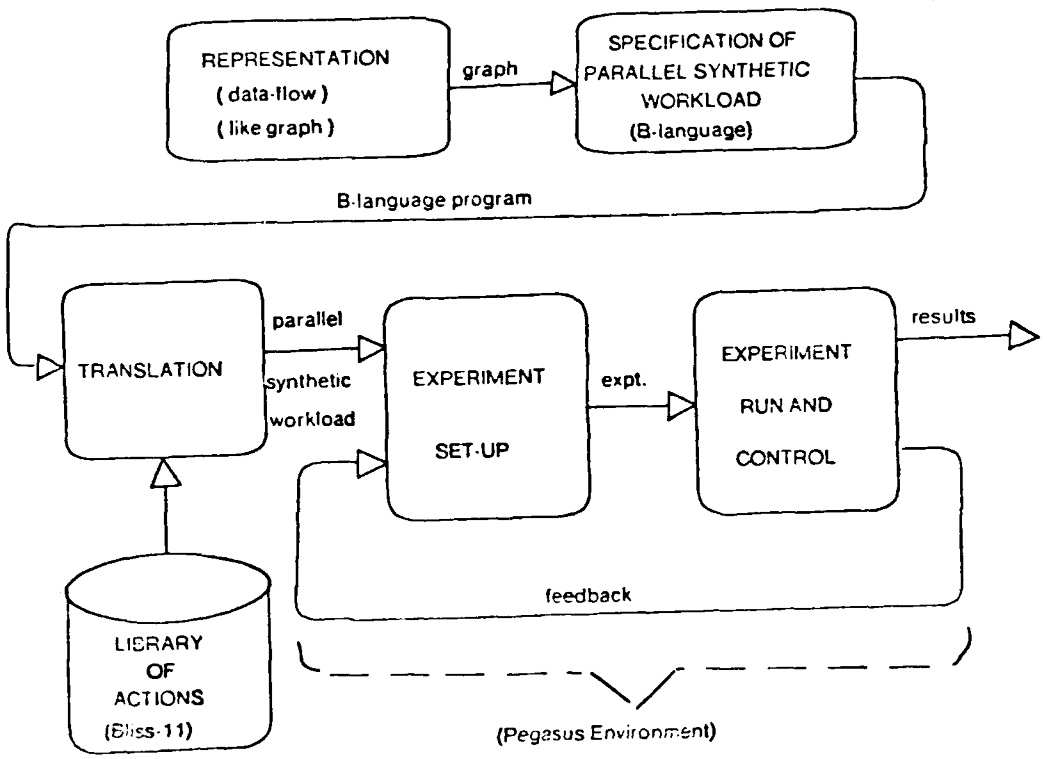


Figure 11: The Components of Pegasus

Figure 11 shows the process of synthetic workload generation for experimental performance evaluation, and its relation to the basic components of Pegasus. A special high level behavior description language, the *B language*, is designed to assist the user to specify his parallel synthetic workload as a directed data flow like graph. Intuitively, synthetic programs are comprised of interesting actions repeated at certain rates. Salient features represented in the B-Language include:

- Processes or *subtasks* are represented as nodes of a graph.
- The flow of control between processes is via messages.
- Buffers allow the buffering of messages, and represent the arcs of the graph.
- Subtasks consist of cyclic *actions* that correspond to the computation to be performed by the corresponding nodes.
- Explicit control of the workload is directly represented via special constructs provided in the language.

An action, or a set of interacting actions, are specified in the B-language from a *Programmable library* of actions. The library is system specific, and is programmed in a systems programming language supported on the machine. Furthermore, the user is given the option of programming his own action or a set of interacting actions and adding them to the library. The B-language assumes the underlying operating system support a message-based communication mechanism. Therefore, many of the language features can be transported to other types of *multiple processor architectures*, including local area networks.

The B-language, along with the library, is translated to generate the user specified Parallel Synthetic Workload on the machine. The generated workload is encased in the *Pegasus Environment*, an interactive environment for controlled experimentation. This Experimentation Environment allows the user to run a set of experiments on the machine with modifications to the workload from one experiment to another. The Environment has two main components. A user interface allowing *interactive* set up experiments. A run-time support environment permits *control* of the workload as specified in the B-language.

4.1.1. Background

Our research vehicle is the Cm* multiprocessor. Cm* is a 50 processor multiprocessor with over 4 million bytes of primary memory designed and implemented at Carnegie-Mellon University. Two operating systems, Medusa and StarOS, have been developed for Cm*. In addition, substantial amounts of utility software built for Cm* runs on other general purpose computers in the department.

Both Medusa and StarOS are general purpose operating systems whose architectures can be characterized as "message based". They are distributed in that each operating system is formed by a collection of parallel processes. Each process, or in some cases groups of processes, is responsible for one operating system function, such as file management. User processes communicate with the operating system processes by sending messages. In contrast to a network where messages must be used for all communication, the processes can directly share memory as well as communicate via messages.

4.1.2. Overview of Paper

Section 2 discusses the graph model representation and its specification in the B-language. Section 3 discusses the concepts of an Experimentation Environment and the extensions to the B-language to support its features. The design of the Experimentation Environment is discussed in Section 4 followed by a discussion of the synthetic workload generation in Section 5. Appendix I discusses the Message Communication Throughput (MCT) experiment to demonstrate the use of Pegasus as an effective tool to parameterize the multiprocessor system. Measurements made with an experimental version on the Medusa operating system are presented.

4.2. Parallel Synthetic Programs -- Representation and Specification

4.2.1. The Representation

The representation of parallel synthetic programs in the B-language is via the graph model of computation. The graph model is one of the popular mathematical tools used to represent and analyze the flow of control and data in parallel programs [12] [26] [3] [23]. For systems exhibiting large-grained parallelism, the model can be used to describe the interaction between parallel executing processes.

It is assumed that the reader is familiar with elementary graph theory. The terms: graph and net; node and vertex; arc, branch, and link, will be considered synonymous. In general, nodes will be images of operators, or computations, and the arc will represent either the flow of data, or control, or both.

A generic parallel program model is defined in [3] as a triple $P = (W, U, C)$ where :

- $W = \{w_1, w_2, ..., w_n\}$, is a set of operators or computations ;
- $U = \{u_1, u_2, ..., u_m\}$, is a set of variables or data ;
- C (to be elaborated), is the control

With each operator or computation is associated an input set $I_i = \{u_{i1}, ..., u_{ik}\}$ and an output set $O_i = \{u_{o1}, ..., u_{op}\}$.

Various graph models have adopted different control (*C*) strategies. The *model adopted in the* B-language to represent parallel synthetic programs is similar to a later version of the UCLA graph model, also referred to as the *complex bigraph* (*complex bi*-logic directed *graph*) [23] [10].

The UCLA graph model [17] defines its control **C** in terms of the vertices of the graph. With each vertex w_i is associated one of the ordered pairs (* , *), (* , +), (+ , *), (+ , +) representing the input and output logic respectively for that node. If the first member of the pair is *, w_i is said to be AND-input logic (otherwise it is of OR-input logic). Similarly, w_i is of AND-output logic (OR-output logic, also called branching vertex) depending on the value of the second element of the pair.

The complex bi-graph extends the UCLA graph by allowing complex (many-to-many, that is multi-head and multi-tail) arcs in the graph [23]. Furthermore, the bigraph also allows *buffering* of control items (also referred to as tokens) and data items on arcs. An additional feature of the bigraph is that each arc is weighted with a number, which represents the number of tokens needed on that arc so that the node can fire.

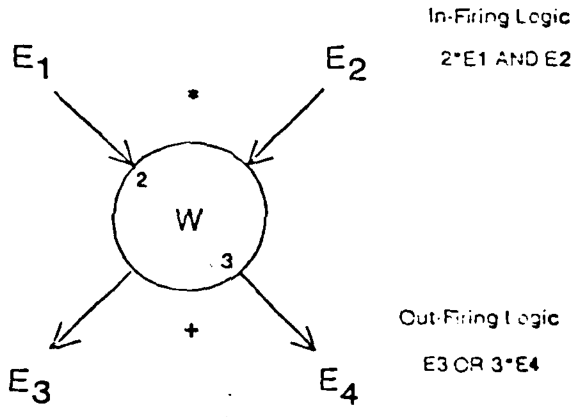


Figure 12: A Bigraph Node

For example in Figure 12, node W is executed if there are two tokens on arc E_1 AND one token on arc E_2. Notice this is a modified variation of the conventional JOIN. Upon completion of execution of the node either one token is placed on arc E_3 OR three tokens are placed on arc E_4.

The B-language represents a parallel synthetic program as a directed data flow graph similar to the complex-bigraph. The node here represents a subtask, or process, that executes in parallel with other subtasks to implement the user specified parallel synthetic program. More formally, the triple represented by the B language is *(S, B, R)*. Observe a one to one correspondence with the parallel model, ***P***, presented on Page 37. For the B-language model we have :

- *S* = $\{s_1, s_2, ..., s_n\}$, is a set of n distinct subtask, or processes, that represent the nodes of a graph and cooperate to complete the user's specified task.

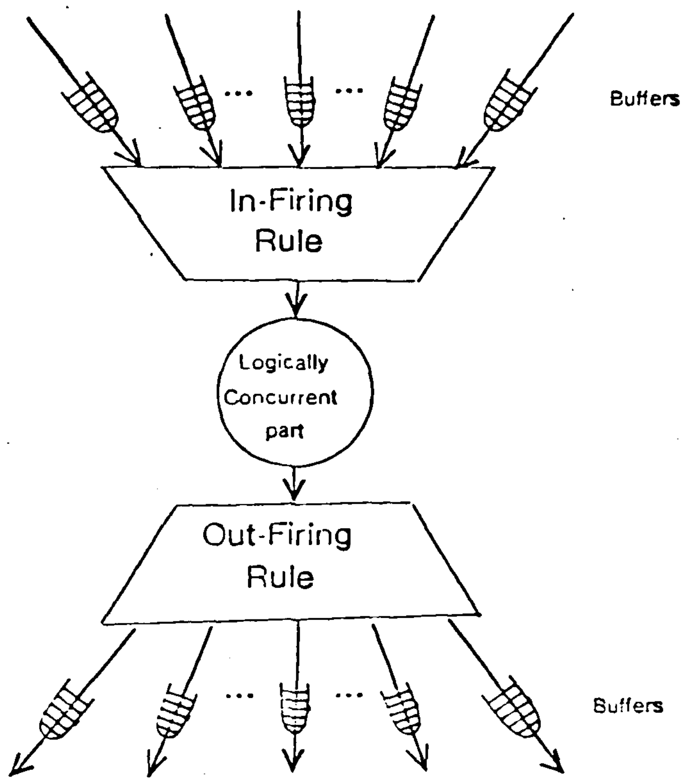


Figure 13: A Generalized Computation Node

- B = {b_1, b_2, ..., b_q}, is the set of q buffers that can queue data variables and control tokens flowing from one subtask to another and represent the arcs of the graph. A semaphore, as will be explained later, is viewed as a special case of buffers. The exact semantics of the buffer is given in Section 4.2.2.1.

- R = {r_1, r_2, ..., r_{2n}}, is the set of firing rules that characterize the precedence relationship between the nodes (subtasks) of the graph. With each node (subtask) there is associated a control tuple (r_{pi}, r_{po}), where :

  - o r_{pi} corresponds to the in-firing rule for subtask s_p.
  - o r_{po} corresponds to the out-firing rule for subtask s_p.

  Observe the similarity of the above control tuple to the ordered pair discussed on Page 38.

The in firing and out firing rules are specified as a logical relationship on buffers only. Note, the semaphore object can be viewed as a special case of a buffer, where a P corresponds to a Receive, and a V corresponds to a Send to the buffer. A generalized computation node, a subtask is, therefore, abstracted as shown in Figure 13.

A multi-headed, multi-tailed control arc (complex arc) is also represented by a single buffer. Consider for example, the graph shown in Figure 14. Notice the complex arc, E_1 is represented by a single buffer B_1. Thus a single buffer, may be present in the in firing and out-firing rules of many subtasks. It may also be

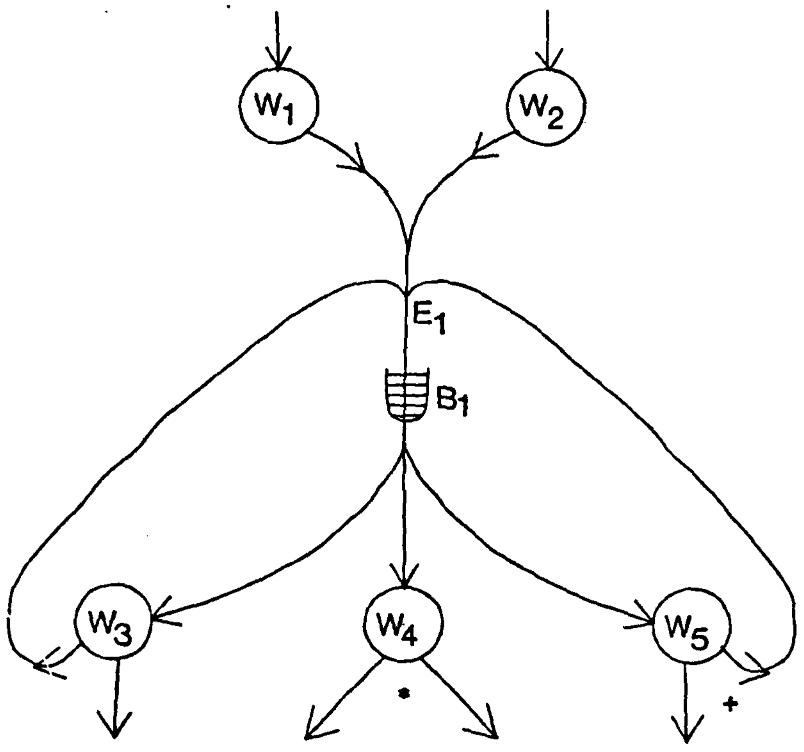


Figure 14: Complex Arc Representation

present in the in-firing and out-firing rule of the same subtask, as buffer B_1 for subtasks W_3 and W_5 in Figure 14.

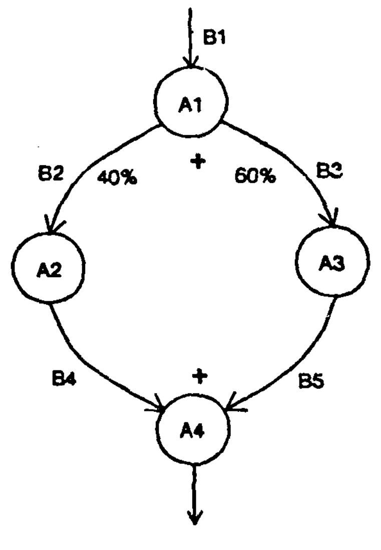


Figure 15: A Parallel Synthetic Program

Thus the B-language represents the interaction of parallel processes via the graph model of computation. A typical example shown in Figure 15 shows a graphical representation of interacting processes in a parallel synthetic program. Subtask A1 is fired by buffer B1 which corresponds to the entry arc of the graph. Upon completion, subtask A1 fires either of subtasks A2 or A3. There is a certain probability associated with the OR-output logic of subtask A1. Finally subtask A4 fires if it receives a token either from A2 or A3. Upon completion, it fires buffer B6 which corresponds to the exit arc of the graph and represents the end of a single execution of the parallel synthetic program. The corresponding subtask declarations in the B-language will be as :

```
SUBTASK A1 {  INLOGIC : B1 ; OUTLOGIC : %40(B2) OR %60(B3) }
           .
           .
SUBTASK A2 {  INLOGIC : B2 ; OUTLOGIC : B4 }
           .
           .
SUBTASK A3 {  INLOGIC : B3 ; OUTLOGIC : B5 }
           .
           .
SUBTASK A4 {  INLOGIC : B4 OR B5 ; OUTLOGIC : B6 }
```

Notice the arcs B_1 to B_6 correspond to distinct buffer names. The delimiter, %, is used to specify the branching probabilities for the arcs of an OR-output logic.

4.2.2. The Specification of a Parallel Synthetic Program

4.2.2.1. Object Specification

The specification of parallel synthetic programs is in the B-language and is based on the object model supported by both operating systems on Cm*. The objects represented directly in the B-language include :

- The *task force* object : The task force abstraction represents a collection of *subtasks* that cooperate to achieve a single logical task. This corresponds to the Medusa *task force* [51] the StarOS *task force* [34] or a *team* of processes in Thoth [11].

- The *subtask* object : This is the basic computation unit that executes in parallel on a processor, and cooperates with similar user-defined objects to compute the overall stipulated multiprocess task. The subtask, therefore, corresponds to the process in StarOS or an activity in Medusa.

- The *buffer* object : The buffer object is a conventional queue of messages and is used by the subtasks to communicate with each other. Generic operations defined on the buffer object include :

  - (conditional / unconditional) Receive a message from a buffer. An unconditional receive on an empty buffer causes the receiving subtask to block. A conditional receive on an empty buffer, on the other hand, does not cause the receiving subtask to block.

- (conditional / unconditional) Send a message to a buffer. A unconditional send on a full buffer causes the sending subtask to block. A conditional send on a full buffer, however, does not cause the sending subtask to block.

- The *semaphore* object : Semaphores synchronize requests for shared resources. Operations on the semaphore include the conventional P and V.

- The *file* object : Files represent a sequence of bytes. Associated with each file is a file pointer which points to the next byte of information to be read or written. Generic file operations include open or close file, and read or write file.

- The *shared* data object : Variables specified in the shared data object are globally shared by all the subtasks of the task force. This allows communication of data and control through shared memory.

- The *table* object : Tables implement functions varying with time. The motivation, and exact semantics of these objects are discussed in section 4.3.

As part of the object specification, particularly for architectures with non-uniform communication costs, it is important to be able to specify the distance between objects. This is particularly important when one wants to estimate the cost of data flow between objects. For example, two subtasks in the same cluster in *Cm*, will experience a lower communication* (data flow) cost then two subtasks which are in different clusters. The relative distance between objects is directly dependent on the absolute location of the objects with respect to the machine. Both StarOS [32] and Medusa have higher level software which permit the specification of inter-object distances (also referred to as proximity relations in StarOS [59], and location specifiers in Medusa). The B-language directly reflects this feature of both operating systems. Examples of some location specifiers for Cm* include: require same Cm, require same cluster, Cm Apart, and cluster apart.

The object specification in the B-language is static. The object type is assumed to be supported directly by the underlying operating system. A minimal set of objects necessary for a message-based system exhibiting large-grained parallelism is identified and represented in the language. The motivation is to maximize the portability of the language design over message-based, object-oriented systems. Special objects, peculiar to a particular operating system, or created dynamically by the runtime system, can be exploited in an indirect fashion via a system specific library of actions. This will be discussed in greater detail in Section 4.3.

4.2.2.2. Programmable library of Actions

The library of actions consists of a collection of system specific, pre-debugged actions, stored as a file in the system data base. The user can selectively include an action, or a set of actions from the library into his B-language program. The actions in the library are programmed in a systems programming language supported on the machine. Examples of pre-defined actions include : send or receive message to or from buffer, input or output to a file, local memory references, block on a semaphore, access to a shared resource.

The user gains flexibility by programming his own special action, or a set of interacting actions, and adding them to the library. The library of actions, in some sense, performs the map from the parallel synthetic program specification onto the underlying multiprocessor system. The B-language thus should be viewed as a portable framework into which system specific actions are inserted from a system specific library of actions.

4.2.2.3. Subtask Specification -- Control Constructs

The B-language control constructs within a subtask are a subset of constructs found in conventional programming language. The motivation is to permit the programming of a sequential set of synthetic actions for each subtask. An enhanced set of constructs could have been incorporated into the language to give the flexibility of a programming language. This option is left open to the user, by allowing him to program new actions into the library in a the systems programming language.

To capture the cyclic nature of synthetic workloads, an action a_k is described in terms of an action-repetition tuple (specified as $<a_i, r_i>$). This implies that action a_i is repeated sequentially r_i number of times, and that constitutes action a_k. An action may be arbitrarily complex, and maybe further composed of action-repetition tuples. Observe, both a_i and the r_i can be parameterized according to the requirements of the synthetic program.

For a subtask the B-language picks a limited subset of the D-structures [14] [37]. The control constructs are :

1. *Basic Actions.* Pre-debugged action, or user-programmed actions, available in the library, are the basic building blocks one may use to construct more complex *actions*.

2. Repetition. An action (either basic or complex) may be repeated a certain number of times, and that, in itself, is another complex action.

3. Compositions. Actions may be composed to form more complex actions. ',' indicates composition. Composed actions may be demarcated into a larger complex action by the delimiters '{' and '}'. For example a complex action may be formed as :

$$
\begin{aligned}
a_{complex} &= a_{composed} : r_{composed} \\
&= \{ \\
&\quad \langle a_i : r_i \rangle, \\
&\quad \langle a_j : r_j \rangle, \\
&\quad \langle a_k : r_k \rangle \\
&\} : r_{composed}
\end{aligned}
$$

Notice, the composed action, $a_{composed}$ is a composition of three actions, each of which is an action-repetition tuple. Also, the composed action forms a part of another action-repetition tuple, which represents $a_{complex}$.

4. Conditional. Of the form **if** p **then** a_i **else** a_j, where the predicate p is restricted to the comparison of integers. The conditional constructs regulate the flow of control, and make the detection of errors possible.

Besides these conventional control constructs, there is a demonstrable need for a non-deterministic construct. This is because the nature of the data for specifying a synthetic workload is usually statistical in nature with some probabilistic flow of control. A multi-way probabilistic branch statement (SELECT) has been introduced. This allows the user to specify multi-way branches, with probabilities associated with each of the branches. An example of the SELECT statement is as :

```
< SELECT {
     600 : < a_i : r_i > ;
     250 : < a_j : r_j > ;
   REMAINING : < a_k : r_k >
     } : 1
>
```

This statement performs action $\langle a_i : r_i \rangle$, 60% of the time, action $\langle a_j : r_j \rangle$, 25% of the time, and action $\langle a_k : r_k \rangle$, the REMAINING 15% of the time.

4.3. The Experimentation Environment

The Experimentation Environment is a hospitable environment which allows the user to run a set of experiments on the machine with modifications to the parallel synthetic program from one experiment to another. This section discusses the functionality of the Experimentation Environment, and the extensions to the B-language which allow a user to utilize this functionality. The basic cycle supported by the environment therefore looks as :

```
WHILE true DO
BEGIN
   SetUp(CurrentExperiment(Feedback(PastExperiment))) ;
   Run(CurrentExperiment) ;
   Make(Measurements) ;
END ;
```

4.3.1. Setting Up Experiments

In setting up experiments the Experimentation Environment has two major functions. First, it should ensure the repeatability and integrity of experiments from one run to another. Second it should allow the user to specify a set of parameters and vary them on a per experiment basis.

4.3.1.1. Repeatability of Experiments

To ensure the integrity and repeatability of experiments it is important that experiments be independent of one another. In other words, it is the responsibility of the Environment to ensure that *side-effects* of one experiment do not perturb the next experiment. For a graph simulation, an example of a side-effect is tokens left over in the edges (buffers) as a result of the previous experiment. In this case, it is the responsibility of the Environment to flush out all buffers after an experimental run.

4.3.1.2. Variation of Parameters

Major modifications in the structure of the parallel synthetic program, without having to re-translate, would pose severe implementation problems. Certain interesting features of the graph structure are suggested as interesting for variation on a per-experiment basis. Some of them have been implemented and found useful in the first implementation of Pegasus.

The variation of the number of active subtasks in the experiment is particularly useful in real-time experimentation, where one wants to determine the number of copies of different subtasks required to meet real time constraints. The language allows for the specification of an array of a particular subtask. In the graph model this corresponds to the specification of multiple executable copies of the same node. The variation of parameters within a subtask allows the user to vary the load generated by the corresponding subtask. An example of the action portion of a subtask is :

```
VARY par3 ;
SUBTASK X { INLOGIC: ABuffer }
  VARY par1, par2 ;
  BEGIN
     < $action1 : par1 >,
     < $action2 : par2 >,
     < $action3 : par3 >
  END
SUBTASK Y { INLOGIC: BBuffer }
  VARY par4, par5 ;
  BEGIN
     < $action4 : par4 >,
     < $action5 : par5 >,
     < $action6 : par3 >
  END
```

The VARY command in the B-language instructs the translator to treat the variables par1, par2, par3, par4, and par5 as special control variables that can be accessed and varied on a per-experiment basis. The language allows two kinds of *vary-variables*, global and local. The global vary-variables are shared across all subtasks in the task force, and allow global control of the task force. On the other hand, the local vary-variables have their scope restricted to the subtask and control the load of the subtask. In the above example, vary-variable par3 is a global vary-variable, and therefore controls the variation of load for both subtasks X and Y. The rest of the vary-variables are either local to subtask X or subtask Y. Notice the vary-variables mainly serve as a function of controlling the experiment, rather than their more standard use in conventional programming languages.

4.3.1.3. Table Objects and the Variation of Parameters in Real Time

An additional important feature of an Experimentation Environment is the variation of parameters as a function of time while an experiment is in progress. This allows a user to impose a set of external time-driven forcing functions onto his workload. The problem of variation of parameters in real-time is only addressed for a subtask. A *table* object is introduced as a feature of the B-language to request the variation of parameters in real-time. The table object provides a table of successive values that the parameter may take with respect to time. Parameters that need to varied in real-time are declared as *runtime-variables* in the subtask. The user can associate a delta-time, and a table object, with each runtime-variable. This forces the runtime-variable to take on successive values from the table after regular time intervals of delta-time. A greater amount of flexibility can be introduced by:

- postponing the binding between the runtime-variable and the table object to as late a stage as possible. A suggested point to establish this binding is on a per-experiment basis.

- permitting the variation of the delta-time of the runtime-variable on a per-experiment basis. This is, in effect, compressing or expanding the time-axis of the corresponding real time function exported by the table object and the associated runtime-variable.

The same runtime-variable may be bound to different table objects on a per-experiment basis. Also two variables may be bound to the same table object for a particular experiment. For example:

```
TABLE ATable { FILE: A.Tab },
      BTable { FILE: B.Tab } ;
              .
SUBTASK A { ...... }
  RUNTIME Var1, Var2 ;
              .
              .
```

The TABLE declaration declares two table objects, ATable and BTable. The FILE parameter specifies the file containing the table values.

4.3.2. Starting and Running Experiments

After having set up his experiment, the user is now ready to start the experiment and run it on the machine. In the graphical representation of the program this corresponds to firing the entry node, that is, placing a token on the entry arc of the graph. To start an experiment, the Environment defines the entry arc of the parallel synthetic program as a system-defined buffer called the $IgnitionBuffer. On the start of the experiment the $IgnitionBuffer is the receiver of free tokens from the Environment. Upon request from the Environment the user may inject an arbitrary number of tokens into the $IgnitionBuffer.The user may now avail of this free source of tokens to start any desired subtask, by specifying the $IgnitionBuffer appropriately in the In-Firing Rule of that subtask. For example :

```
SUBTASK X { INLOGIC : $IgnitionBuffer }
                    .
                    .
SUBTASK Y { INLOGIC : $IgnitionBuffer }
                    .
                    .
SUBTASK Z { INLOGIC : ($IgnitionBuffer) OR (ABuffer) }
```

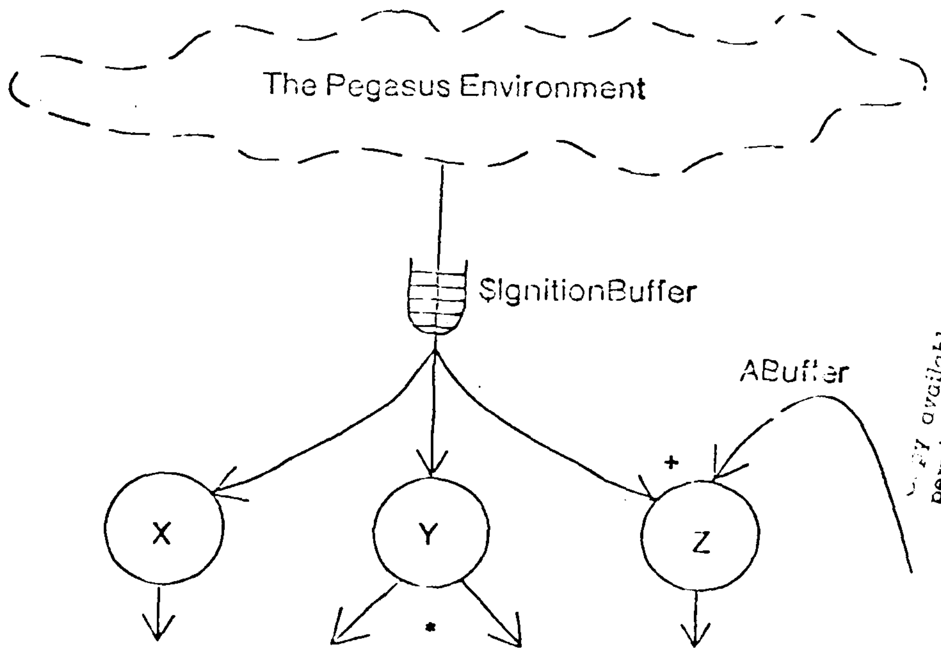


Figure 16: The $IgnitionBuffer as a Complex Arc

The three subtasks, X, Y, and Z, have their in-firing rules specified as a function of the $IgnitionBuffer (Figure 16). Thus successive runs of the experiment can be started by firing the $IgnitionBuffer the appropriate number of times. The concept is, in some sense, analogous to the ignition spark plug of an internal combustion engine, where the engine, in this case, is the task force.

Similar to the concept of the $IgnitionBuffer, the Environment defines a $TerminationBuffer which corresponds to the exit arc of the parallel synthetic program and marks the end of a particular experimental run. It is the responsibility of the user to fire the $TerminationBuffer, thereby informing the Environment about the end of an experiment.

4.3.2.1. External Event Generation

In message-based systems, a typical external event in a parallel program is the event associated with the arrival of a message. Thus a reasonable requirement of the external event generator is to be able to deliver token messages at pre-specified buffers, according to pre-specified functions of time. Table objects are used to implement the functions of time. A typical B-language command declaring a message event is:

```
MSGEVENT AEvent = ABuffer @ ATable ;
```

This command requests the message-event generator to deliver messages to ABuffer with inter-message time-periods as specified by successive entries of ATable. The user is permitted to associate a different table object with the msgevent-variable on a per experiment basis.

4.3.3. Timing Measurements

As part of the Experimentation Environment, the B-language permits the user to embed some basic timing mechanisms into his load, thereby allowing him to make some monitor-independent measurements. The TIME command declares a set of parameters which can be used to measure times taken for various actions in the code for a subtask.

To make more detailed measurements, it is necessary for the Environment to interface with a monitor. In the current design of Pegasus, the Environment interfaces with Simon [69], a distributed relational monitor written for object-oriented systems. The parallel synthetic program in Pegasus, for most purposes, behaves like any other program running on the system and monitored by Simon. However, the higher level of abstraction presented by B-language, permits special higher level interaction with the monitor thereby permitting a greater deal of flexibility.

4.4. Design of the Experimentation Environment

This section briefly develops a design for the Experimentation Environment and its relationship to the parallel synthetic workload. The focus is on the modular decomposition of the design and the inter-module relationships (interfaces).

4.4.1. Modular Decomposition

Broadly speaking, the executable program generated by the translator can be decomposed into two logical parts :

1. The executable Load Program module.
2. The supporting Environment module.

The executable Load Program module corresponds to the user specified parallel synthetic workload. The possible functionality that can be exported by the executable Load Program includes :

1. Performing timing measurements, as specified by the TIME command in the user specification (Section 4.3.3). This permits the user to measure the time taken for the specified action-repetition tuple.
2. Implementing the user specified parallel synthetic workload, that is, the interactions between subtasks as specified by the graph model, and the synthetic program executed by each subtask.
3. Implementing the enabling and disabling of the subtask for a particular experimental run.

The Environment module exports most of the functionality of the Experimentation Environment. The objective in the design is to minimize any perturbation that it may introduce into the experiment. This module can be further decomposed into the following sub-modules (Figure 17) :

1. The User Interface is a conventional command interpreter that provides a graceful interface with the user on a per experiment basis. Depending on the user command, the User Interface communicates and controls the Environment submodules and the Load Program module. The user can now set-up, control, and run experiments interactively on the machine. The User Interface module has access to symbol table information for the parameters that need to be varied on a per experiment basis. Appendix II lists some of the commands of the User Interface.
2. The Message Event Generator permits the generation of external stimulus for the executing Load Program. The Message Event Generator is designed as a separate process which vvdelivers token messages at pre-specified buffers according at pre-specified functions of time. As a special case the time interval between successive deliveries of the token may be a constant.
3. The Driver for RUNTIME Variables : The primary function of the RUNTIME Variable Driver module is to force all RUNTIME variables to vary as a function of time as specified by its associated table object and delta time. The RUNTIME variable takes on successive values from the table after regular time intervals of delta-time.
4. The Clock Module permits access and manipulation of a clock.
5. The Random Number Generator supports the execution of the probabilistic SELECT

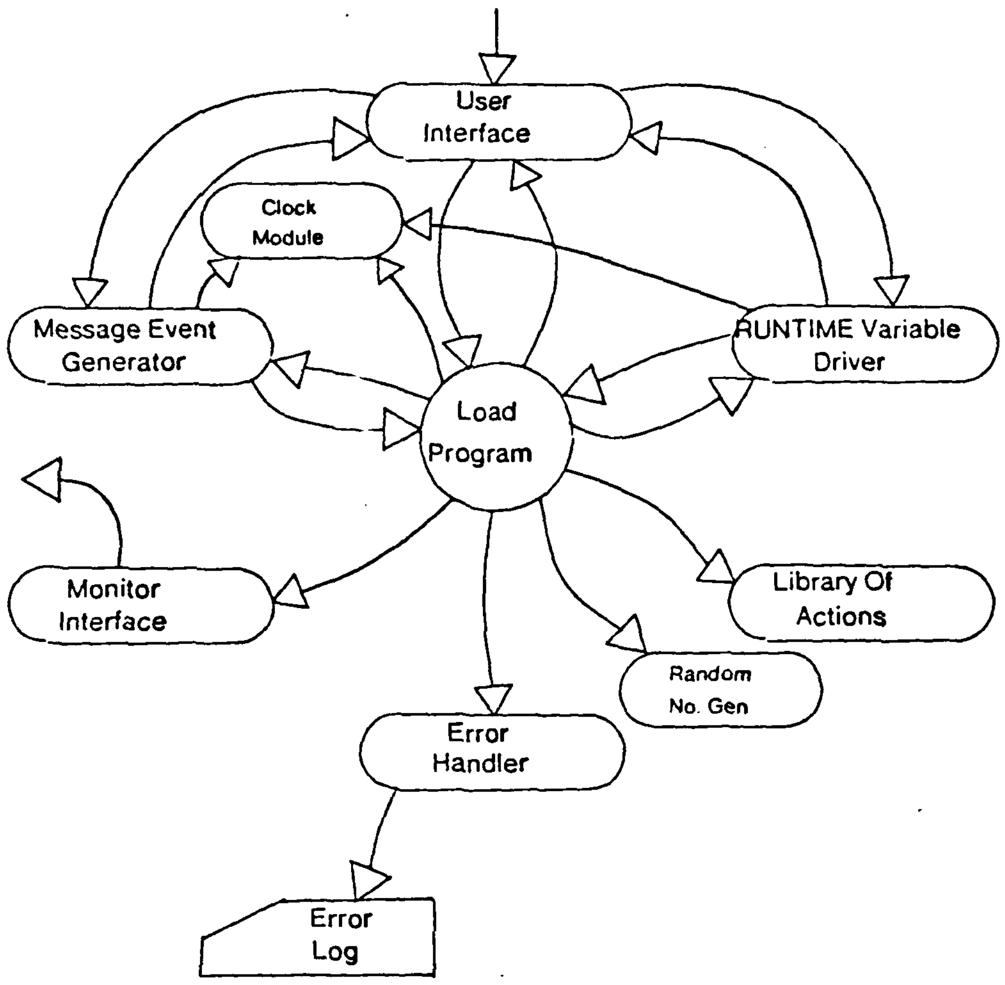


Figure 17: Detailed Decomposition

statement in the language.

6. The Monitor Interface allows additional monitoring of the Load Program.

7. The Error Handler and Log permits the logging of errors that the operating system reports.

4.5. Design of the Generation Mechanism on Cm*

One of the objectives of the design of the B language was to present a conceptually unified machine to a novice user. As a result the language was designed to express both the interaction of parallel subtasks, and the synthetic program executed by each subtask. The user now has to specify only one file corresponding to his load program. Figure 18 shows how a typical B-language program gets translated in the implementation on Cm*. A typical B-language program consists of two parts : the object and global declarations, and the subtask declaration. The translator takes this program as input and generates a set of Bliss-11/StarOS or Bliss-11/Medusa files, a definitions file (.DFS), and a linker command file. Each subtask generates a corresponding Bliss-11 module. Symbol table information for parameters local to the

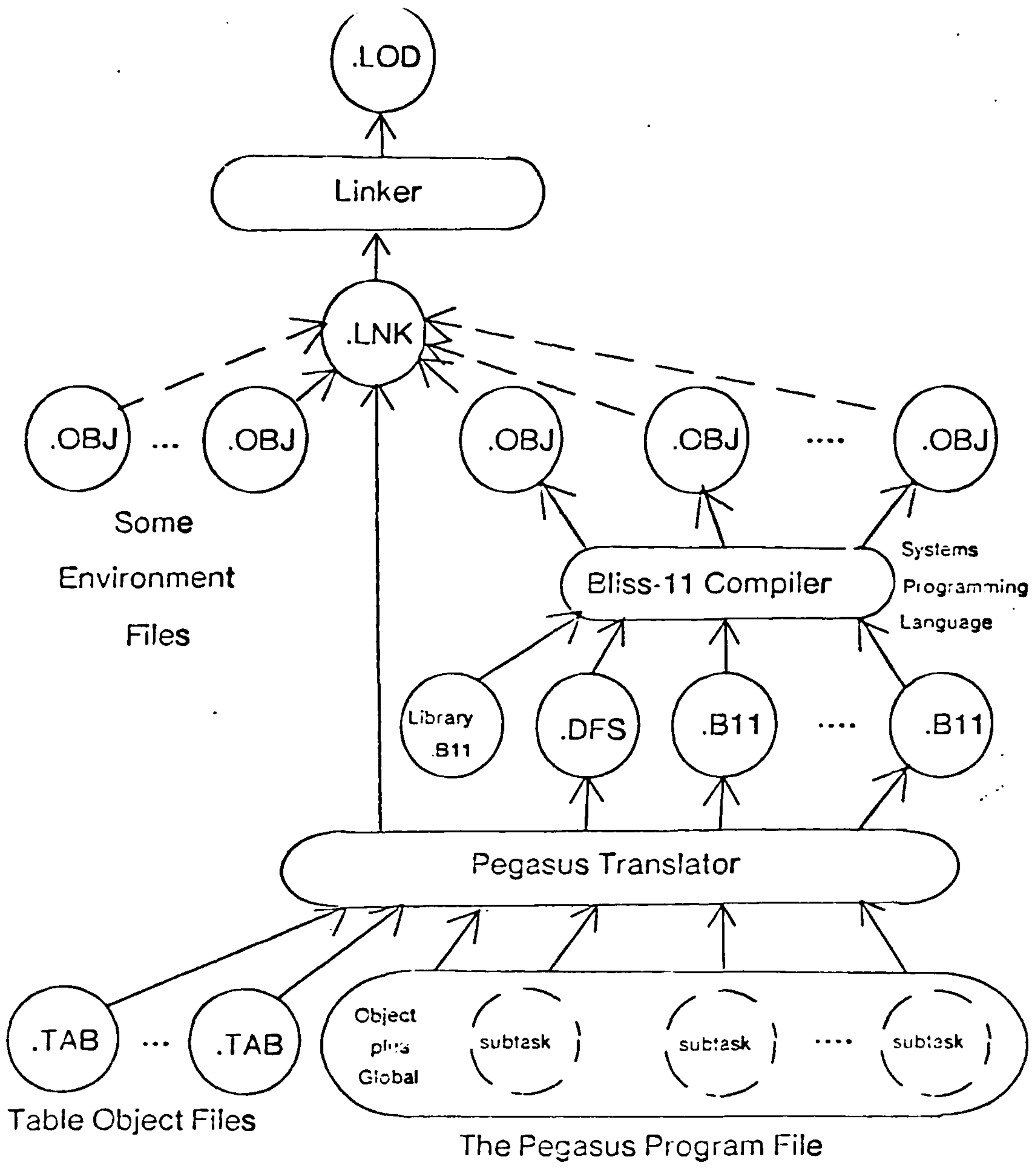


Figure 18: The Flow of Files

subtask are demarcated as CSECTS [7] in the corresponding Bliss-11 module. Besides a Bliss-11 module per subtask, the Pegasus translator also generates a global definitions file (.DFS), which along with the library file is compiled with each Bliss-11 module. Furthermore, to extract the symbol table information for global parameters, such as global VARY-variables and MSGEVENT variables, another Bliss-11 module is created as a separate file with the appropriate information encoded in CSECTS.

The most vital part of the translation process is the generation of the linker command file for the parallel synthetic program. This file describes the structure of the parallel synthetic program, its objects, and their accessibility to the parallel subtasks. The symbol table information is extracted as CSECTS and wired in by convention, so that it is accessible to the appropriate processes. The linked file is now ready for loading on the machine.

4.6. Conclusion

The process of experimental performance evaluation has been simplified by providing a tool for the representation, specification, and execution, of parallel synthetic programs in a controlled environment. Special care has been taken in the design to ensure portability over message-based, object-oriented systems. The minimal set of objects necessary for a message-based system exhibiting large-grained parallelism have been identified and represented in the language. Within a subtask the control constructs have been kept simple. The current design has, for the most part, been implemented on the Medusa operating system. Another version of the design is being implemented on the StarOS operating system. The initial use of Pegasus has been mainly as an experimentation tool for parameterizing and comparing the features of Medusa and StarOS. The issue of generating workloads representative of actual multiprocessor applications has not yet been examined. This is primarily because of the difficulty encountered in deriving the workload characteristics for such applications.

Future extensions to Pegasus include the design of a fault inserter, and the specification of redundant software structures for reliability studies.

4.7. Acknowledgements

The authors would like to acknowledge the contributions of some of the concepts and the implementation by the other members of the Multiprocessor Performance Evaluation Group: Xavier Castillo, Robert Chansler, Ivor Durham, Peter Highnam, Anita Jones, Daniel Siewiorek, and Pradeep Sindhu.

Appendix I Experimentation

The primary objective of this section is to demonstrate the usability of Pegasus. The Message Communication Throughput (MCT) experiment tries to characterize the message communication bandwidth for a message based system as a function of some specified parameters. For the Medusa operating system the task force scenario is a Sender subtask and a Receiver subtask communicating via a buffer. In this case, the buffer corresponds to the pipe object of Medusa, through which messages can be sent by value. The parameters of the experiment are :

- Size of a message, B bytes.
- Usage of the received message, f(B). Where, f(B) can be either Log(B), or O(B), or O(B^2), or O(exp(B))[1].
- The physical distance between the Sender and the Receiver. The current experiment has been run for the inter-cluster case.

For this simple experiment, it is assumed that the Sender and the Receiver subtasks are the only subtasks scheduled on their processors. Also, it is required that the Sender and the Receiver subtasks be coded such that the Sender be able to send an infinite number of messages to the Receiver. Furthermore, it is a assumed that the Send and Receive operations are unconditional, that is, the Sender blocks if it sends to a full buffer, and the Receiver blocks if it receives from an empty buffer.

The user specification of the experiment as a parallel synthetic program is as :

```
TASKFORCE MCTExpt

BUFFER CBuffer { SIZE: 1024 } ;

VARY NoOfIterations, SizeOfMessage ;

SUBTASK Sender { INLOGIC:  $IgnitionBuffer ,
                 LOCATION: Cm1-1, Require same cluster }
LOCAL PBlock[3], SrcObject[1024] ;
BEGIN
  < $SetSendPBlock(PBlock, SizeOfMessage, SrcObject, CBuffer) : 1 >,
  {
    < $SetUp(SrcObject) : SizeOfMessage >,
    < $Send(PBlock) : 1 >,
  } : NoOfIterations
END
```

[1]This is the Order notation

```
SUBTASK Receiver { INLOGIC:  $IgnitionBuffer ,
                   LOCATION: Cm2-1, Require same cluster }
LOCAL PBlock[3], DestObject[1024], CopyObject[1024] ;
VARY UsageOfMessage ;
BEGIN
  < $SetRecvPBlock(PBlock, SizeOfMessage, DestObject, CBuffer) : 1 >,
  {
    < $Receive(PBlock) : 1 >,
    < $CopyMessage(DestObject, CopyObject) : UsageOfMessage >
  } : NoOfIterations
END
```

The Message Communication Throughput is defined as :

```
                2*SizeOfMessage * NoOfIterations
      MCT =  ----------------------------------   bytes/sec
               T(StartSender) - T(FinishReceiver)
```

The Experimentation Environment automatically computes the values of the start and the end times of each subtask of the parallel program. Thus the values of T(StartSender) and T(FinishReceiver) can be queried via the User Interface. Observe, for every send-receive transaction, the act of setting up the source object ($SetUp(SrcObject, SizeOfMessage)) by the Sender subtask defines the number of words (SizeOfMessage) that it desires to communicate with the Receiver Subtask.

I.1. MCT -- Measurements

The MCT experiment was run for combinations of the parameters: size of message, and usage of message. Figure 19 shows a plot of MCT versus message size for varying usage patterns. The results show some of the tradeoffs involved of sending large and small messages on the Medusa virtual machine. The curves indicate that the overall time to send N messages of size M each is of the form:

```
  [K1*N + K2]*M   microseconds
```

Here K2 reflects some constant overhead required to set up the message channel, and K1 is the average cost of sending one word. The MCT is therefore proportional to :

```
  (M*N)/([K1*N + K2]*M) = 1/(K1 + K2/N)   bytes/sec.
```

The above expression approaches 1/K1 for large N, and approaches 0 for small N. In this case we have :

```
  1/K1 = (appx) 45 K-Bytes/Sec
or  K1 = 18.2 microseconds / byte
```

Figure 19 therefore characterizes the message throughput of the Medusa message mechanism. The use of the Pegsus as a tool to perform this experiment saved a substantial amount of time. In particular,

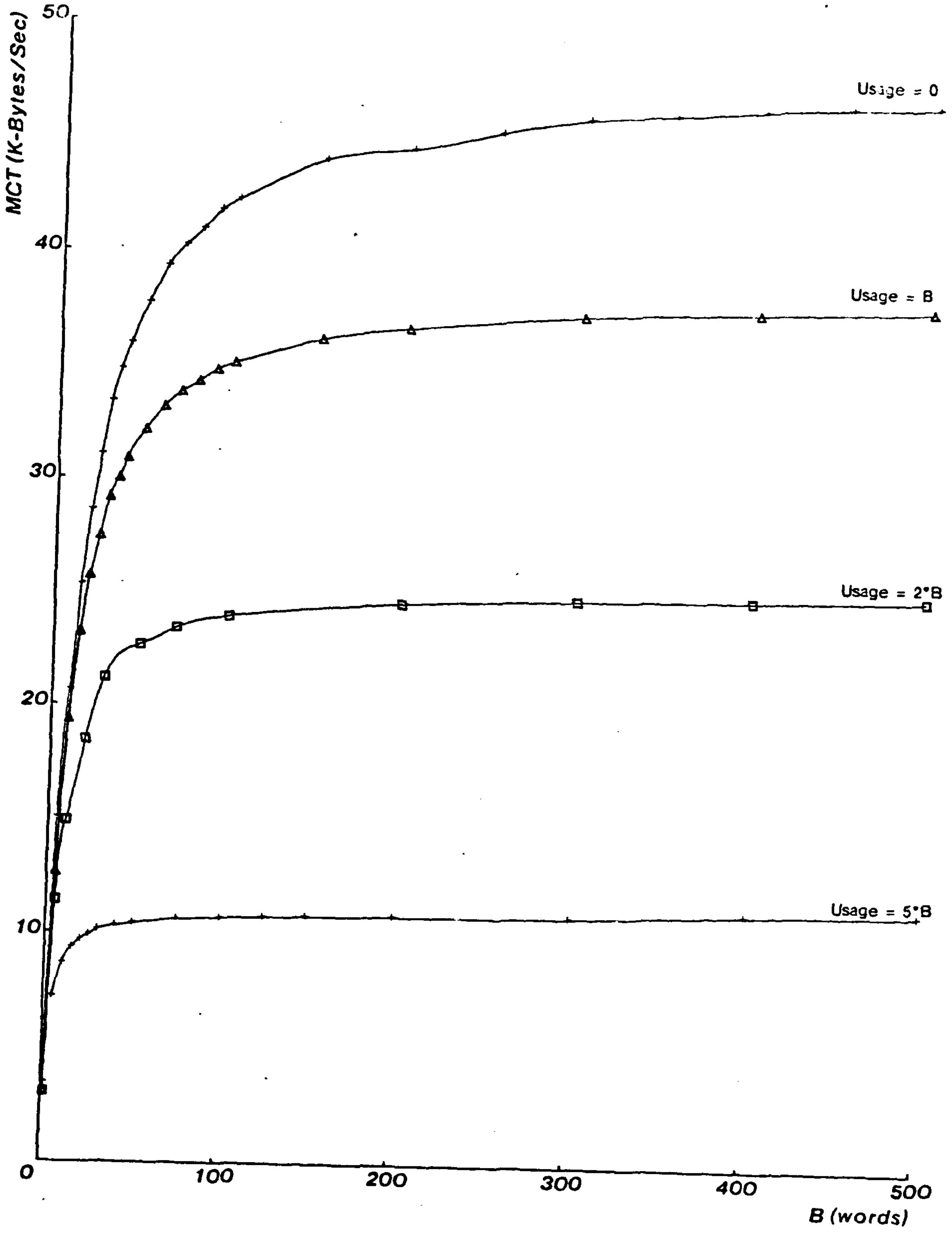


Figure 19: MCT versus Message Size for Varying Usage Patterns

with the AUTOMATIC command of the User Interface (see appendix for details) these results were acquired in a short amount of time.

Appendix II
User Interface Commands

II.1. VARY / QUERY / HOWLONG Command

The VARY command allows the user to modify the VARY parameters on a per experiment basis. The QUERY command allows the user to query the value of a VARY parameter of a specific subtask. The HOWLONG command returns the value of a TIME parameter for the specified subtask. A typical user interaction for the VARY command is as (defaults are given in square brackets) :

```
-PEGASUS- vary
   Name Of Subtask ==> ASubtask
   Name Of Parameter [All] ==> ?

    *** The Following are the VARY parameters ***
        AParam
        Bparam

   Name Of Parameter [All] ==> AParam
   Value -- (BASE 10) ==> 20.
-PEGASUS-
```

II.2. AUTOMATIC Command

The AUTOMATIC command puts the user into Automatic Experimentation mode, where the User Interface commands, and values of parameters, are read in from a command file, and a series of experiments performed. By giving appropriate instructions to the monitor, the user can also automate the data collection for his set of experiments. The objectives of this command are two fold :

1. To speed up any routine experimental work assuming that the user has already decided the parameters of his experiment.

2. To speed up the exploration of entire regions in the experimental space of the user.

A typical user interaction is as follows :

```
-PEGASUS- automatic
  Name of Command File ==> /usr/as/acmd
-PEGASUS-     .
              .
  ( input is read from Command File and echoed on terminal )
              .
              .
```

II.3. FIRE Command

The FIRE command fires enables all enabled subtasks, and then fires a specified number of tokens into the $IgnitionBuffer. A typical user interaction is as follows (defaults are given in square brackets) :

```
-PEGASUS- fire
   How many Ignition Tokens  ?? [1] ==> <CR>
   Start the Event Generator ?? [N] ==> <CR>
-PEGASUS-
```

If the user desires to start the Message Event Generator, the FIRE command communicates appropriately with the Message Event Generator module. The pseudo code for the command is as follows :

```
BEGIN
  FOR AllUserSubtasks DO
  BEGIN
     IF (Subtask Enabled) THEN
        Enable(Subtask)
  END ;
  Number = AskUser("How many Ignition Tokens ") ;
  IF (AskUser("Start Event Generator") = True) THEN
     Signal(StartEventGenerator) ;
  ClockReset ;
  INCR I FROM 1 TO Number DO
     Fire($IgnitionBuffer) ;
  IF (AutomaticMode = True) THEN      ! in AUTOMATIC mode
     Block(on $TerminationBuffer) ;   ! waits for the end of
END ;                                 ! the experiment.
```

II.4. MSGEVENT / GENSTOP Command

The MSGEVENT command permits the user to vary the time interval between successive messages to the buffers associated with a MSGEVENT variable. The variation of the time interval can be either by associating a constant with the MSGEVENT variable or by associating a table object with it. In the latter case the Message Event Generator fires off the specified buffer at time intervals that are successive values of the table. A typical user interaction for a constant inter-message time interval is as :

```
-PEGASUS- msgevent
   Name Of Message-Event ==> AEvent
   Name Of Table Object  ==> constant
   Value -- (BASE 10)    ==> 20.

-PEGASUS-
```

The GENSTOP command permits the user to stop the message event generator and thus control the duration of his external stimulus in the experiment.

II.5. STATUS / TOTIME Command

The STATUS command gives the status of the various subtasks in the user task force. Status information includes information such as whether the subtask is running, what is its location with respect to the hardware, and other operating system specific information. The TOTIME command gives the user access to the start time, the end time, and the total time taken by the latest execution of all subtasks. A typical user interaction is as follows :

```
-PEGASUS- totime
                              (       TIME -- Minutes : Seconds        )
 Subtask ID      Subtask       Start Time       End Time       Total Time
 ---------------------------------------------------------------------------
     1          ASubtask      00 : 00.002     00 : 12.346     00 : 12.344
     2          BSubtask      00 : 02.879     00 : 13.999     00 : 11.120

-PEGASUS-
```

The STATUS, TOTIME, and GENSTOP commands can be used while an experiment is in progress. This is unlike the rest of the commands which are recommended for use on a per-experiment basis.

II.6. ENABLE / DISABLE / TFINIT Command

The ENABLE / DISABLE commands permit the user to enable or disable a particular subtask. Disabling a subtask essentially makes the subtask unavailable for the current experimental run. This allows experimentation with a different number of servers for a node in a given execution of a graph. A typical user interaction is as follows:

```
-PEGASUS- disable
   Name Of Subtask ==> CSubtask
   Name Of Subtask ==> BSubtask
   Name Of Subtask ==> ↑G
-PEGASUS-
```

The TFINIT command initializes the task force for the next experimental run.

II.7. RUNTIME Command

The RUNTIME command allows the user to associate a given table object and a delta-time with the specified RUNTIME variable (Section 4.3.1.3). A typical user interaction is as :

```
-PEGASUS- runtime
  Name of Subtask  ==> ASubtask
  Name of RUNTIME  variable ==> AVar
  Value of Delta-Time (milliseconds) [175] ==> 100.
  Name of Associated Table Object [BTable] ==> ATable
-PEGASUS-
```

Appendix III
The B-language Grammar

This appendix describes the B-language grammar in reasonable detail. The objective is to give the user a formal introduction to the language. The description, however, is not exhaustive.

```
<taskforce>
    ::= TASKFORCE <taskforce← name>;<object← declaration>
<object← declaration>
    ::= <passive← objects>;<global← declarations>;<active← objects>.

<passive← objects>
    ::= <buffer← object>
    ::= <file← object>
    ::= <semaphore← object>
    ::= <table← object>
    ::= <shared← object>
<buffer← object>
    ::= BUFFER <buffer← declaration>;
<buffer← declaration>
    ::= <buffer← attributes>,<buffer← declaration>
    ::= <buffer← attributes>
<buffer← attributes>
    ::= <buffer← name> {SIZE: <integer>}
<buffer← name>
    ::= <name>
    ::= <name>[1 .. <integer>]
<file← object>
    ::= FILE <file← declaration>;
<file← declaration>
    ::= <file← attributes>,<file← declaration>
    ::= <file← attributes>
<file← attributes>
    ::= <file← name>{ <file← OSname> }
<semaphore← object>
    ::= SEMAPHORE <semaphore← declaration>;
<semaphore← declaration>
    ::= <semaphore← attributes>,<semaphore← declaration>
    ::= <semaphore← attributes>
<semaphore← attributes>
    ::= <semaphore← name> { INITIAL: <integer> }
<table← object>
    ::= TABLE <table← declaration>;
<table← declaration>
    ::= <table← attributes>,<table← declaration>
    ::= <table← attributes>
<table← attributes>
    ::= <table← name> { FILE: <file← host← name> }
<shared← object>
    ::= SHARED <shared← declaration>;
<shared← declaration>
    ::= <shared← attributes>,<shared← declaration>
    ::= <shared← attributes>
<shared← attributes>
```

```
    ::= <shared← variables> { LOCATION: <location← specifier> };
<shared← variables>
    ::= <variable>,<shared← variables>
    ::= <variable>

<global← declarations>
    ::= <bind← declaration>
    ::= <msgevent← declaration>
    ::= <vary← declaration>
<bind← declaration>
    ::= BIND <bind← relation>;
<bind← relation>
    ::= <binding>,<bind← relation>
<binding>
    ::= <variable> = <conatant>
<msgevent← declaration>
    ::= MSGEVENT <msgevent← relation>;
<msgevent← relation>
    ::= <event← binding>,<msgevent← relation>
    ::= <event← binding>
<event← binding>
    ::= <buffer← name> @ <time← period>
<time← period>
    ::= <table← name>
    ::= <constant>
<vary← declaration>
    ::= VARY <vary← variables>;
<vary← variables>
    ::= <variable>,<vary← variable>
    ::= <variable>

<active← objects>
    ::= <subtask← object><active← object>
    ::= <subtask← object>
<subtask← object>
    ::= SUBTASK <subtask← name>{ <subtask← TFlevel> } <subtask← block>
<subtask← name>
    ::= <name>
    ::= <name>[1 .. <integer>]
<subtask← TFlevel>
    ::= INLOGIC : <inlogic← experssion>
    ::= OUTLOGIC: <outlogic← expression>
    ::= LOCATION: <proximity← relation>
    ::=
<subtask← block>
    ::= <subtask← declaration><subtask← body>
<subtask← declaration>
    ::= <vary← declaration>
    ::= <runtime← declaration>
    ::= <time← declaration>
    ::= <local← declaration>
<runtime← declaration>
    ::= RUNTIME <runtime← variables>
<runtime← variables>
    ::= <variable>,<runtime← variables>
    ::= <variable>
<time← declaration>
    ::= TIME <time← variables>
<time← variables>
    ::= <t← variable>,<time← variables>
    ::= <t← variable>
<t← variable>
    ::= <variable>
<local← declaration>
    ::= LOCAL <local← variables>
<local← variables>
    ::= <variable>,<local← variable>
    ::= <variable>
<subtask← body>
    ::= <initial← action><service← loop>
<initial← action>
    ::= <action>
<service← loop>
    ::= BEGIN <action> END
<action>
    ::= <delim← begin><action> : <repetition><delim← end>
    ::= IF <boolean← expression> THEN <action> ELSE <action>
    ::= <compound← action>
    ::= <library← action>
<compound← action>
    ::= { <compose← action> }
<compose← action>
    ::= <action>,<compose← action>
    ::= <action>
<repetition>
    ::= <constant>
    ::= <variable>
<delim← begin>
    ::= <
    ::= LOOP(<t-variable>)
<delim← end>
    ::= >
    ::= ENDLOOP
```

5. The Cm* Test Bed

Edward F. Gehringer, Anita K. Jones, Zary Z. Segall

5.1. Introduction

Interest in multiprocessor architecture has grown steadily over the past ten to fifteen years. VLSI technology offers the potential for building multiprocessors which are substantially larger than present computers to solve problems that cannot be solved today. A substantial number of multiprocessor designs exist, yet only a few multiprocessors have been built with a high degree of parallelism, say thirty or more processors.

Although multiprocessors appear to have cost/performance and reliability benefits, the computing community has relatively little experience in the actual use of multiprocessors. Consequently, little is known about how well the potential of multiprocessors can be realized in practice. This is exactly the thrust of the Cm* research project. This paper presents our experience in two important aspects of multiprocessing: first, the structure and organization of a multiprocessor test bed, and second, a collection of experimental results acquired from it.

A distributed system can be seen as being composed of an architectural component and a behavioral component. The first component consists of hardware, firmware, and software elements, and the relationships between them. The behavioral component is characterized by the way the architecture acts in the presence of a workload. The architectural component should be flexible, and the behavioral component should provide a controllable and measurable behavior.

These two goals have been emphasized in the Cm* test bed. Hardware-architecture flexibility has been realized by a programmable interconnection network. The two operating systems StarOS and Medusa provide adaptable mechanisms and policies for running experiments with application programs. The workload, the measurement tools, and the experimentation control are integrated into an experimentation environment. This environment complements the other support programs such as compilers and loaders with facilities for the specification, monitoring, and analysis of experiments.

The Cm* project has successfully constructed a 50-processor multiprocessor and two operating systems, thus demonstrating the feasibility of several aspects of multiprocessing. This paper explores some of those aspects, beginning with the Cm* hardware. Next, it focuses on the software—operating systems, utilities, and the higher-level software used to control experiments and gather data from them. Section 5.3 uses these experimental results to investigate how well various algorithm structures exploit the

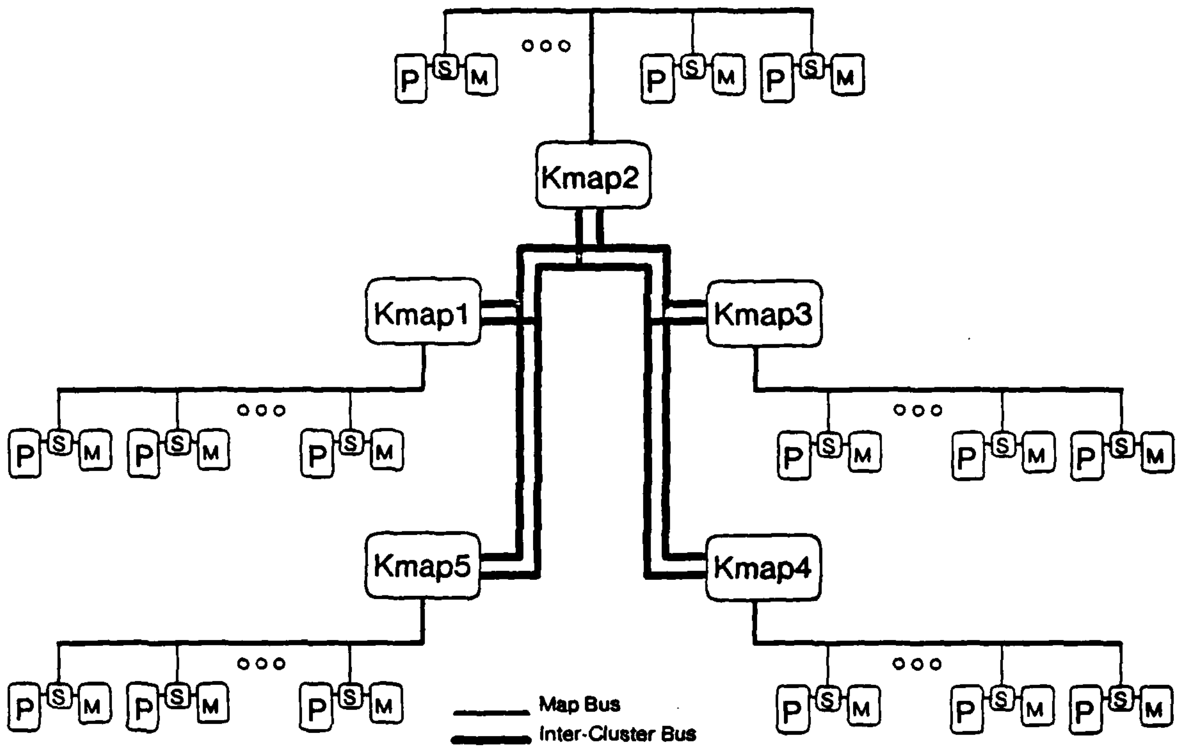


Figure 20: Five-Cluster Cm* Configuration

potential parallelism of Cm*. Finally, the results of extended measurements on the hardware itself are presented.

5.2. The Cm* Experimentation Environment

5.2.1. The Cm* Hardware

The Cm* hardware consists of a number (currently 50) processor-memory pairs, called *Cm's*, connected by a *hierarchical, distributed switching structure.* Cm* is partitioned into sevei :l (currently 5) *clusters* of up to 14 Cm's each; the Cm's in an individual cluster are connected via a *map bus* to a mapping processor, called a *Kmap*, through which they communicate with each other (Figure 20). The clusters themselves are connected via *intercluster busses*. For a Cm to communicate with a Cm in a *remote* cluster, it first sends a request to the Kmap in its cluster, which forwards it to the Kmap in the remote cluster, which in turn passes the request to the target Cm.

Consequently, any Cm can reference memory anywhere in the system; the memory appears to be a single large memory. Nevertheless, the communication paths are of different length and induce a performance hierarchy: references by a Cm to its local memory take about 3 μsec.; references to another Cm in the same cluster require 8.6 μsec.; and intercluster references take about 35.3 μsec. [31].

Each Cm is a Digital Equipment LSI-11 with 64K or 128K bytes of memory, modified by the addition of a local switch, or *Slocal*, which examines each memory reference generated by the processor. Using its internal mapping tables, the Slocal decides whether the reference is to the processor's local memory (in which case the reference takes no longer than a memory reference by an unmodified LSI-11), and if not, passes the reference on to the Kmap. The Kmap is a high-speed microprogrammable communication controller with a cycle time of 157 nanoseconds, a control store of 4K 80-bit words and a data RAM of 4K 80-bit words. Since the Kmap is programmable, it is possible to experiment with different processor-memory and interprocessor communication strategies. Further, the border between software and firmware is fluid, and can be moved to investigate different operating-system implementations. For example, important operating-system functions can be placed in the Kmap to improve performance or to protect them from user programs.

Each Kmap has two bi-directional ports which may each be connected to a separate intercluster bus. This permits the clusters to be connected in many different ways. At present, there are two intercluster busses to which all five Kmaps are attached; hence there is a direct path between each pair of Kmaps, and messages need never be forwarded via intermediate Kmaps. All communication between Kmaps and from Kmap to Cm is performed by packet switching rather than circuit switching to avoid deadlock over dedicated switching paths. The Kmap is provided with eight process-state *contexts* so that it can service up to eight requests concurrently by switching from one context to another. Typically, it switches away from a context while awaiting completion of a memory access emanating from that context.

A major advantage of the Cm* structure is its extensibility. Because the switching structure is not centralized, it does not grow more complicated as the number of processors is increased. By contrast, C.mmp [83], an earlier multiprocessor constructed at CMU, employed a 16-by-16 crosspoint switch to route memory references to its sixteen processors. The complexity of such a switch grows by a factor of n^2 as n, the number of processors, is increased. Cm* is built from clusters of limited size, which may be interconnected arbitrarily. Thus there is no inherent limit to the number of processors or the amount of shared memory. Because intercluster messages need not pass through a central switch or bus, there is also no architectural limit to the overall intercommunication bandwidth. Indeed, it has been estimated [74] that a Cm* structure of up to 10,000 processors could be built.

5.2.2. Flexibility of the Switching Network

Multiple-processor computer systems vary along a continuum from shared-memory multiple-ALU machines like the ILLIAC IV [5], to multiprocessors in which each processor may access any portion of memory though it is directly connected to one portion of memory, to computer networks where each processor may directly address only its own memory. Cm*'s Kmaps can be microprogrammed to emulate either of the latter two structures, or other interconnection structures such as rings and cubes.

For example, if the Kmap is loaded with microcode which cannot map an address generated by one processor to the memory of another, Cm* becomes, in effect, a network of LSI-11's. It is useful to compare the performance of a multiprocessor with a network, as it seems intuitively true that a network should be cheaper than a multiprocessor with the same processor-memory pairs, due to the relative simplicity of the interconnection hardware.

Levy Raskin [55] performed two experiments comparing Cm* as a multiprocessor with Cm* as a network. His results showed that, at least for some practical multiprocessing algorithms, a network can be competitive with a multiprocessor. The first experiment used an integer-programming application which exhibited rather low communication traffic:

> *Set-Partitioning Integer Programming* The set-partitioning algorithm implemented for Cm* uses an enumeration algorithm which performs an n-ary tree search in a large, relatively sparse binary matrix for a minimum-cost solution. The matrix is two dimensional; its size is usually on the order of hundreds by thousands. The problem is to solve
>
> $$\min(\underline{c}\cdot\underline{x} \rfloor\ A\,\underline{x} = \underline{e},\ x_j = 0 \text{ or } 1 \text{ for } 0 \leq j \leq N)$$
>
> where A is an $M \times N$ binary matrix, $\underline{c}$ is a vector of length N, and e is the identity vector of length M.
>
> As an example, consider the airline-crew scheduling problem. The rows of the A matrix correspond to a set of flight legs to be covered during a specified period, and the columns of A correspond to a possible sequence of tours of flight legs made by one crew; $\underline{c}$ is a vector containing the cost of each tour. A feasible solution consists of a set of tours that satisfy all the flight legs (one and only one crew makes a flight leg). The algorithm seeks the solution with the lowest cost.

The integer-programming algorithm was run on both Cm* configurations. In the network configuration, all interprocessor communication must be via messages (transmitted by value) rather than shared memory. The shortest message could be transmitted in 85 μsec. The algorithm was tested on five sets of data. When the matrix and other read-only global variables were replicated in each processor, the communication overhead was moderate, about 1.3 Mbits/second. Nonetheless, the network configuration

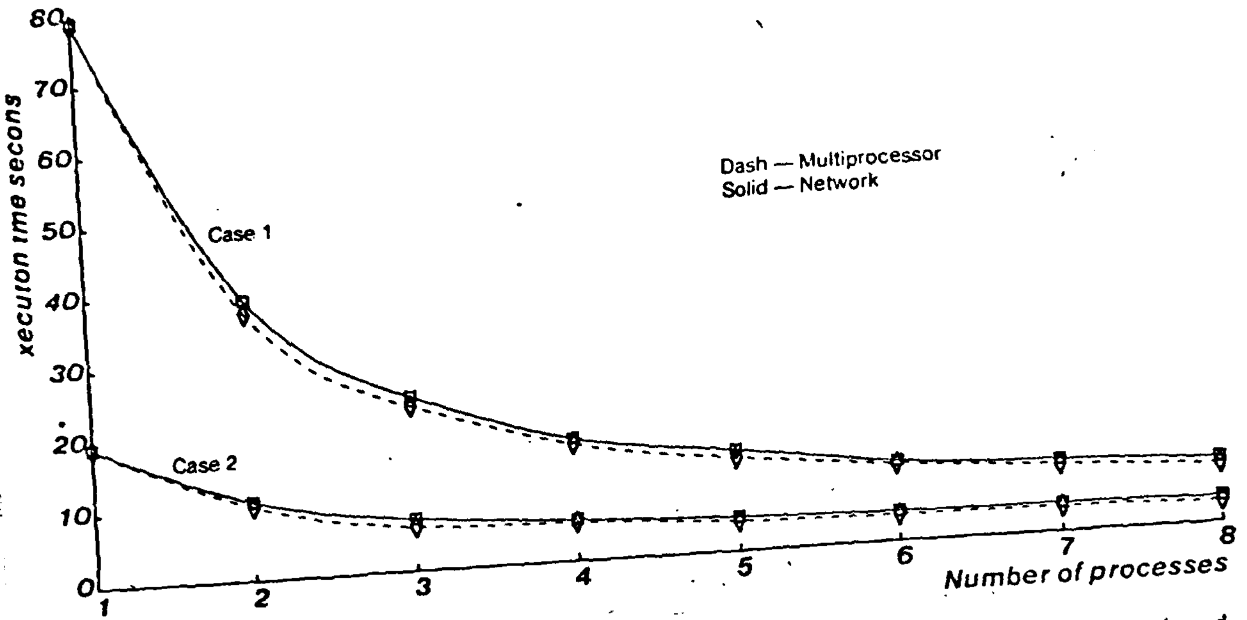


Figure 21: Comparison of Integer Programming on Network and Multiprocessor Configurations, Cases 1 and 2

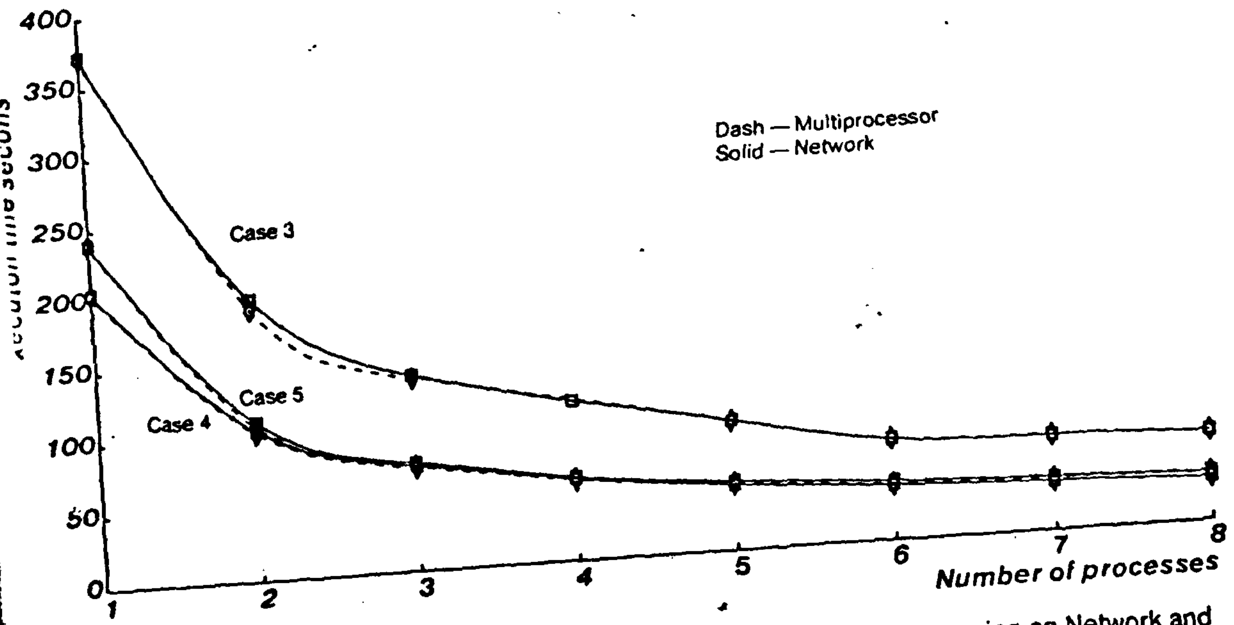


Figure 22: Comparison of Integer Programming on Network and Multiprocessor Configurations, Cases 3 through 5

actually performed better than the multiprocessor configuration (Figures 21 and 22), perhaps because its microcode is simpler.

The second network experiment involved a speech-recognition task with complex and intense communication demands. It was an application of HARPY, a speech-recognition system developed at CMU [39, 18]. It implemented a voice-input "desk calculator" with a 32-word command vocabulary. Despite the fact that the algorithm was carefully tuned to minimize interprocessor communication, the optimal number of processors in the network configuration was only three; as additional processors were added, execution time increased due to the high message rate. By contrast, the multiprocessor configuration could achieve performance gains with up to seven processors.

To implement large programs on Cm*, it is necessary to expand the LSI-11's sixteen-bit address space. Here, too, the Cm* architecture provides considerable latitude. Each Slocal has one *map bit* and one relocation register for each 4K-byte *page* of its Cm's address space. The map bits can be selectively set by the Kmap to cause various portions of the Cm's address space to be mapped, via the Kmap, to remote Cm's or remote clusters. References which are not mapped to other Cm's can be relocated within the local memory of the same Cm. The Slocal relocation registers too can be updated by the Kmap microcode. Hence a running program can be given access to different portions of Cm* memory at different times. The Kmap's data RAM can be used to cache recently accessed mapping information. Different Kmap microcodes have employed very different strategies for mapping addresses and managing the cache.

5.2.3. External Control of Experiments on Cm*

There are several reasons why Cm* needs to communicate with other computers. When multiple groups are using Cm* simultaneously for operating-system development, some external machine must keep track of who has what resources so that the groups do not interfere with each other. When large multiprocessor experiments are performed, perturbation is minimized if performance data is archived by an external computer. It is occasionally desirable to perform an application cooperatively between Cm* and other computers. Finally, as a research computer, Cm* does not warrant a fully developed set of utilities. Software development can be streamlined if the compilers and editors on other machines can conveniently operate on Cm* code.

Some of these functions are provided by a PDP 11/10 known as the *Cm* Host*. Anyone who wishes access to Cm* first logs into the Host from a terminal anywhere in the Computer Science Department. Then he reserves some set of resources. These usually consist of individual Cm's, or entire clusters. To

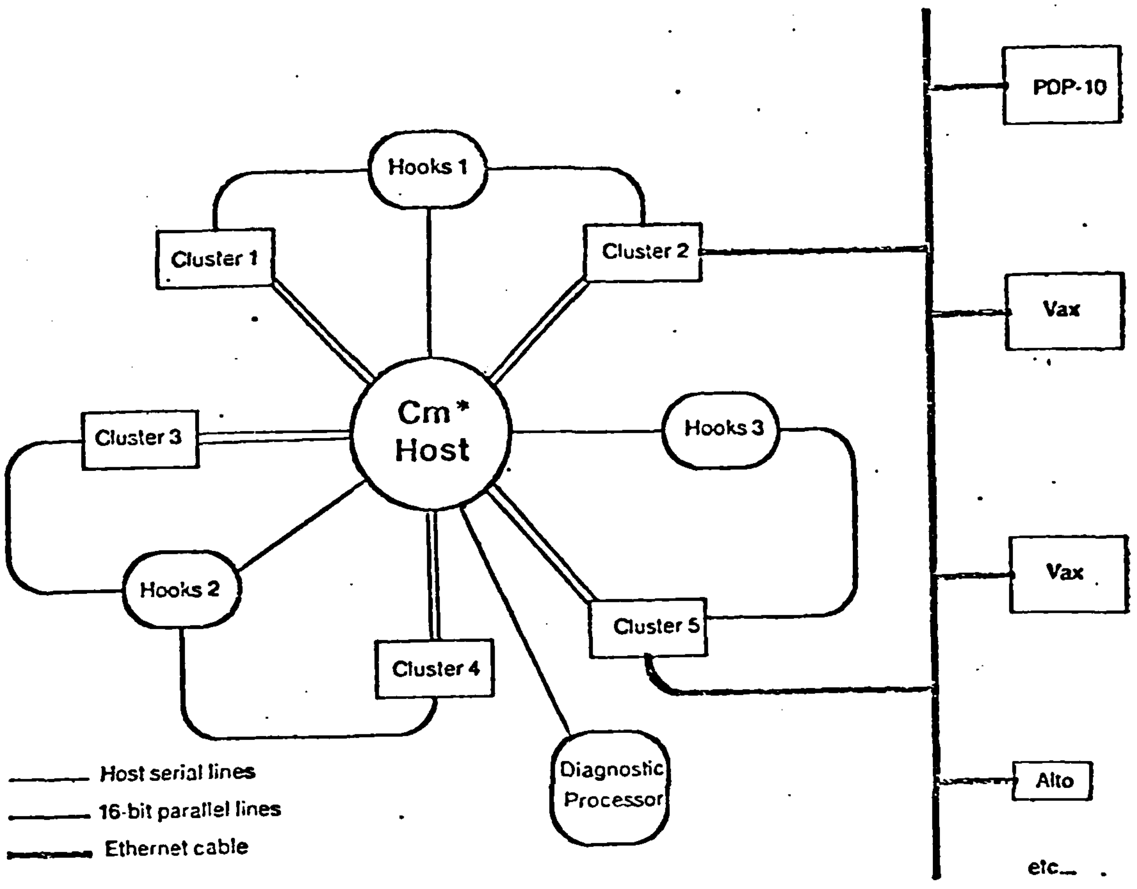


Figure 23: The Cm* Hardware Environment

use a pre-loaded Cm* operating system, the user need reserve only a single serial line attached to one Cm. To load an operating system, one or more entire clusters are required. During the daytime, two different operating systems are normally running in two different partitions of Cm*.

The Host can be used to start, stop, and single-cycle Cm's. It can also be used to "listen to" serial lines selectively, so that if several processors are sending him output simultaneously, the user can temporarily "gag" all but one of them to look at output from the other uninterrupted. Should he choose to listen to all his processors, he can have each line of output prefixed by the name of the processor which sent it.

Through the Host, one gains access to the three *Hooks processors*, which are special microprocessors attached to the Kmaps. The Hooks run a powerful debugging tool known as *KDP*, which can load, start, and stop and single-step a Kmap, and examine its registers or RAMS. With KDP, microcode can be debugged on Cm* itself, rather than on a software simulator. This is important, because in a multiprocessor, microcode bugs are often timing dependent and incapable of re-creation in a simulator.

The *diagnostic processor* is a PDP 11/10 connected to the Host. It runs diagnostic programs on unused Cm's and to maintain an *error log*. These records are important for the management of a large multiprocessor.

Cm* communicates with other general-purpose computers over the Computer Science Department's ETHERNET. Files, particularly object-code files, are transferred over this link to Cm*. The data-aggregation facilities of the *SIMON* monitor [70] run on a Vax and gather data which is shipped in packets from the resident monitor running on Cm*. Commands are sent to the resident monitor from the Vax, making it possible to control Cm* experiments from an external machine. Figure 23 is a diagram showing how Cm* is interfaced to other devices and computers.

5.2.4. Experimentation-Environment Software

Cm* provides a flexible, layered experimentation environment. The experimenter may use as many of the layers as he deems appropriate for his application—from the "bare" hardware to sophisticated experiment-management tools. We shall mention the components of this environment, then return to describe them in detail.

An operating system is the foundation of any software environment. Cm* has two: STAROS and MEDUSA. They are instrumented with sensors to allow monitoring. The utilities run as user programs on top of an operating system. On Cm* they are heavily oriented toward providing facilities for creating and controlling sets of concurrent processes. The top level of the experimentation environment consists of three components, a schema manager, a monitor, and a workload generator. These allow speedy construction of synthetic tasks to exercise various portions of the multiprocessor.

5.2.4.1. Operating Systems

STAROS [29] is an operating system for the support of *task forces*: collections of processes cooperating toward the achievement of a single goal. These processes are usually small, with less responsibility than the average process which runs on a uniprocessor, and they proceed in parallel to take advantage of the multiprocessor architecture. STAROS strives to make processes cheap enough that many functions which are ordinarily accomplished by procedure calls—requests for operating-system services such as memory allocation, for example—can be performed instead by separate concurrent processes.

In addition, STAROS is an *object-oriented* operating system: every action performed by the system is the application of some function to one or more objects. STAROS objects include basic objects, which are similar to segments in ordinary virtual-memory systems; stacks, deques, and mailboxes, whose semantics are implemented in microcode; and abstract objects of user-defined types. The representation (e.g. data) of an abstract object is protected from being accessed or modified except by the small set of procedures which constitute its *type manager*. This facilitates the construction of software in terms of modules which have small and well protected interfaces.

Finally, STAROS is a *message-based* operating system. Message primitives, which are efficiently implemented in Kmap microcode, furnish support for interprocess communication and synchronization and function invocation.

MEDUSA [50], another Cm* operating system, has been strongly influenced by the underlying Cm* hardware. The goal of the MEDUSA project is to create an operating system with three attributes: *modularity, robustness*, and *performance*. Two aspects of the architecture influence how these goals are attained: Cm*'s memory is uniformly addressable—processes on different processors can reference the same memory; and the hardware is distributed, with a time penalty for accessing distant words of memory.

An examination of these issues resulted in an operating system with three properties. Functions of the operating system are performed by disjoint utilities that are physically distributed; each utility implements some abstraction for the rest of the system. Utilities are organized as task forces, which are similar to STAROS task forces, except for the degree of sharing: memory may not be shared among an arbitrary set of processes; rather each segment is either global to the task force, or private to a particular *activity*—the MEDUSA term for a component process of a task force. Third, utilities communicate via messages, as in STAROS.

As in STAROS, the MEDUSA message operations are microcoded for the sake of efficiency. The main difference is that MEDUSA messages are byte streams sent through pipes, similar to pipes in UNIX. A pipe can be nearly 4K bytes long, so a rather long message can be sent, providing there is room in the pipe. On the other hand, STAROS messages consist of only one word, or one *capability* (a protected pointer to an object). Consequently, STAROS message communication is generally done by reference—a pointer to the data is passed—while MEDUSA messages are passed by value.

In order to promote an efficient implementation, MEDUSA has attempted to adhere more closely to the Cm* hardware structure than has STAROS. For example, STAROS supports a full capability-based addressing scheme. An address names a capability, which indirects through a descriptor to a segment of memory. A MEDUSA address names a descriptor directly, avoiding one level of indirection, but affording less flexibility in sharing.

Many of the differences between the designs of the two systems can be explained as follows. StarOS views Cm* basically as a multiprocessor. Task forces consists of processes that share global data. Hence it is convenient and efficient for processes to communicate by passing *references*. All physical movement of data is explicitly requested by some process. Processes may migrate from one processor to another.

Medusa views Cm* basically as a tightly coupled network. Messages are passed by value. Transmission of a message physically relocates that message from one memory to another as in a network. Block movement of data is integrated into the message primitives. As on a network, a process remains associated with a specific processor. Recall, however, that Medusa does permit processes in the same task force to share memory independent of its physical location.

5.2.4.2. Support Software

Few existing languages offer comprehensive facilities for defining the many components—code, data, processes, and mailboxes—that constitute a task force. Objects must be created and named, and placed in appropriate physical memories. Both operating systems have provided support software to deal with these issues. Medusa has built its own linker, known as Medlink. Its directives allow the task-force author to control how resources are utilized by the activities that make up a task force.

The StarOS project has developed a separate language for the specification of task forces. Called Task [33], it provides inter-process naming and resource-allocation mechanisms. It can be thought of as an extension to the Bliss language [84], in which StarOS modules are written. Task provides a way to declare names which can be shared among the StarOS modules in a task force. It generates loader directives for constructing the task force. Task can specify the objects and processes which are to be created when a task force begins execution. Its resource-usage directives can state, for example, that a process should execute on the same Cm which contains its code; that two processes should share the same Cm; or that two other processes should execute on different Cm's or in different clusters—to decrease contention, for instance.

5.2.4.3. Support for Experimentation

Several elements make up Cm*'s *Integrated Instrumentation Environment* (IIE).

- First there is the software which *manages* the experiment—takes a description of the experiment, causes the experiment to be run, and then stores its results. This is called the *schema manager.*

- Second, the Simon monitor [70] gathers resource-utilization data while the experiment is running. It gets its information from sensors embedded in the operating system. Each sensor records the state of a resource.

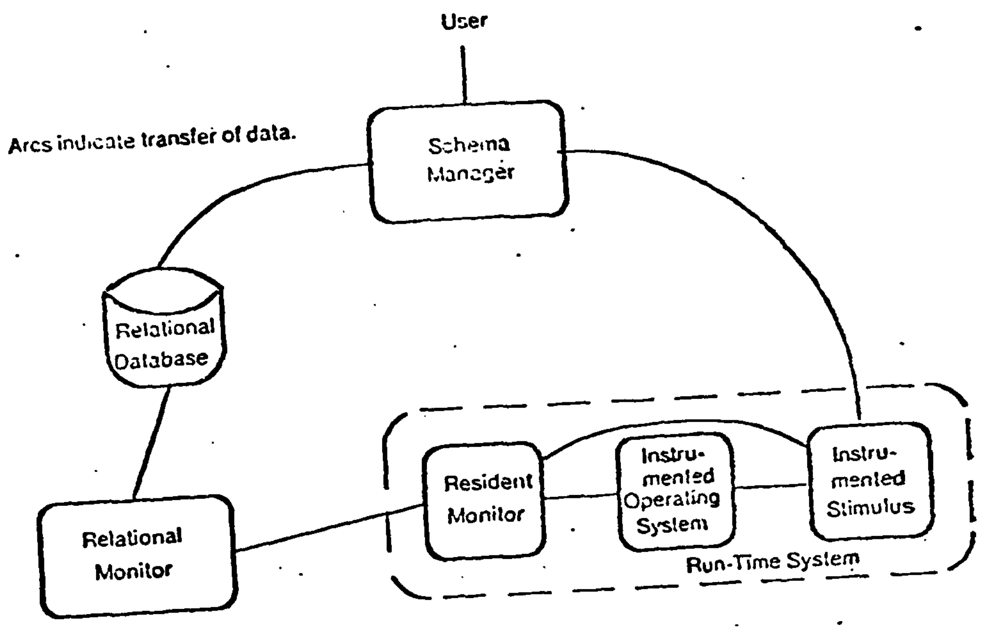


Figure 24: The Integrated Instrumentation Environment

- Finally, there is the task force which is to be measured, called the *stimulus*. It may be either an application program or a *synthetically generated workload*. *Cm*'s workload generator* is PEGASUS [61].

Figure 24 shows how these components fit together. Active at experiment run time are the instrumented stimulus; the instrumented operating system; and the resident monitor, a process which enables and disables the sensors and collects data from them.

A *schema* is a complete experiment description, consisting of a task force to be measured, monitoring directives, system configuration information, and experiment directives written in the workload-generator language. The results of an individual experiment are captured in a *schema instance*. Schemata are archived by the *schema manager*. They may be created either from scratch with a conventional text editor, or generated automatically during a series of runs, by having the schema manager record the commands which are used. The schema manager scans the schema and sends directives to the run-time system. These include global initialization commands for the entire experiment along with commands to set up, start, and terminate each run.

SIMON is a high-level multiprocessor monitoring program. The user asks a high-level question, which SIMON attempts to map into low-level questions. For example, if asked for the utilization of a particular *processor*, SIMON would consult its relational database to discover where the resource was located, and what kind of sensor existed to record usage of the resource. Then it would enable the sensor. Some of

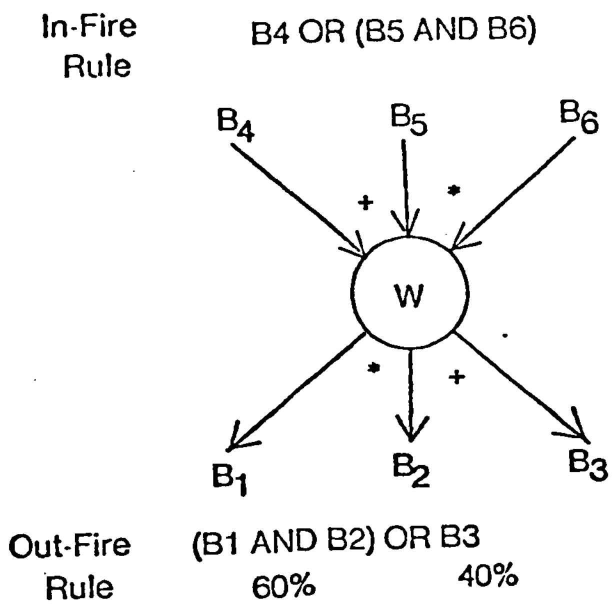


Figure 25: A Node Corresponding to a Subtask

the entries in the database tables contain information which does not vary much with time, for example, the list of processors in the system. Other entries instead contain descriptors for sensors embedded into the operating system. If one of these entries is read, the corresponding sensor will be activated. SIMON itself runs on a Vax and communicates with the resident monitor on Cm* under STAROS. A compatible resident monitor will soon be built for MEDUSA.

The final component of the IIE is the workload generator PEGASUS. Some measurements can best be made with synthetic workloads which exert precise loads on particular components of the system. In addition, it is usually quicker to compose directives for a workload generator than to write an application program with the same characteristics.

PEGASUS represents a task force as a graph. The nodes of the graph are the subtasks (i.e. processes or activities), and the arcs are buffers (e.g. mailboxes) that can queue data variables flowing from one subtask to another. The firing rules are boolean expressions that state which arcs must contain tokens (input firing rule) before the node fires and places tokens on which output arcs (output firing rule). The output firing rule can be expressed probabilistically. For example, it can state that an output token will be placed on arc *B3* with a probability of 40%, or on both arcs *B1* and *B2* with a probability of 40% (see Figure 25).

The subtasks themselves are programmed in terms of simple statements and control constructs. A simple statement says that a particular action is performed a given number of times. For example, $\langle a_j, r_j \rangle$ says that action a_j is performed r_j times. An action consists of a memory reference or the firing of a token. Simple statments may be arranged sequentially, or they may be encapsulated in loops. The *Select* statement allows one of the branches of a case-like statement to be chosen probabilistically. There is another statement which varies the values of parameters each time a loop is performed. This allows an experiment to be repeated automatically under slightly different conditions each time. PEGASUS has been interfaced to MEDUSA, and a STAROS implementation is in progress.

5.3. Research in Distributed Systems

A multiprocessor like Cm* is an versatile test bed. Algorithms can be tested to see how efficiently they decompose for multiprocessing. Interprocess communication systems—such as the STAROS and MEDUSA message systems—can be compared. Experiments can be performed with different ways of organizing task forces. The hardware can even be evaluated, to decide which components are performance bottlenecks and deserving of close attention in future multiprocessor designs. We will describe some of the results which have been obtained using Cm*.

One important question is whether multiprocessors can be cost-effective computational engines; in other words, is there a substantial amount of computing that they can perform more efficiently than alternative architectures—say, fast uniprocessors or networks of smaller computers? Germane to this question is the issue of *speedup*.

5.3.1. Speedup

At first glance, it might seem that most algorithms would run faster on a multiprocessor. Few algorithms, after all, are strictly sequential by nature. But offset against the potential parallelism is the overhead of creating, synchronizing, and communicating with additional processes. These are pitfalls on which many algorithms falter. The question thus becomes: Are there algorithms for important classes of problems which can effectively exploit the parallelism offered by a multiprocessor?

Speedup is a useful measure of whether an algorithm succeeds in harnessing the potential parallelism of a computer like Cm*. It is defined as E_u / E_m, the ratio of the elapsed time required by a uniprocessor algorithm to the elapsed time taken by a multiprocessor algorithm for the same computation. If n processors are used, the speedup is generally between 1 and n. Sometimes when a multiprocessor algorithm is run on a one-processor configuration, the speedup will be less than one; an expressly uniprocessor algorithm would have been faster. Occasionally one encounters a speedup of more than n. We take up this topic in Section 5.3.1.5. For most algorithms, the speedup curve is convex in the number of processors; speedup rises as processors are added, up to a certain point. Then the speedup begins to fall, meaning that adding more processors actually slows down the computation. Thus, another interesting question is, What is the optimal number of processors for executing each algorithm?

During the past few years, we have measured the speedup of several algorithms on Cm*, under widely varying conditions. Early experiments were performed with only a 10-processor Cm* system, precluding realistic multicluster investigation. In later experiments, the number of Cm's per cluster has changed from time to time. Some experiments have enjoyed the full support of an operating system; others have been built from scratch, using only rudimentary Kmap microcode. Among these algorithms are a quicksort, a partial differential equation solver, a railway-network simulation, and the integer-programming algorithm first mentioned in Section 5.2.2. We shall describe them as we consider particular speedup issues.

5.3.1.1. Theoretical and Practical Speedup

Some algorithms lend themselves to more speedup than others. We say that the algorithm exhibits a *linear* speedup if the time taken by an n-processor version is one-nth the time required by the uniprocessor version. Two of our algorithms, partial differential equations and integer programming, have a theoretical possibility of nearly linear speedup. On the other hand, the theoretical speedup s of the quicksort algorithm is only

$$\frac{1}{s} = \frac{1}{p} + \frac{2 - \frac{\log_2 p}{p} - \frac{2}{p}}{\log_2 n}, \tag{1}$$

where p is the number of processes and n is the number of elements sorted.

Linear speedup is the maximum normally obtainable by multiprocessor algorithms. Computations which fail to attain it suffer a *decomposition penalty* [75], which has two components, algorithm penalty and implementation penalty.

The *algorithm penalty* arises from the nature of the algorithm itself, and it also has two components [9]. One of these is the synchronization penalty imposed by the algorithm; in other words, the amount of time that some processors are idle while waiting for other processors to deliver results. The other is the complexity of the reconstitution computations; in other words, the time required to combine the results generated by the individual processors into an overall result for the entire computation. This component is insignificant in the algorithms presented in this paper, but not for all algorithms, for example, the power-systems simulation implemented on Cm* by Dugan, Durham, and Carey [15, 16, 9, 76].

The *implementation penalty* is a consequence of executing the algorithm on a particular hardware and system-software configuration. It is also made up of two components. The first is the degree of coupling between the parallel processors. It reflects the overhead of the interprocess communication (e.g. via global data or message passing) required by the algorithm. This penalty diminishes, for example, as the message-communication microcode is optimized. The other component is the impact of process and data placement: Cm*'s memory-reference hierarchy slows down processes which need to reference non-local data.

Algorithms with little decomposition penalty show the most speedup. This implies little synchronization or communication, few references to large global data structures, and simple reconstitution computations: in short, processes which are relatively small and independent.

5.3.1.2. The Influence of Synchronization

How much does synchronization cost? As a case in point, consider the partial differential equations algorithm:

> *Partial Differential Equations.* The objective is to solve Laplace's partial differential equation (PDE) with given boundary conditions (Dirichlet's problem) by the method of *finite differences.* The equation
>
> $$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = 0 \qquad (2)$$
>
> is solved for points of an *m*-by-*n* rectangular grid, where only the values at the outer edges of the grid are given. The solution is found iteratively. On each iteration the new value of every element is set to the arithmetic mean of the values of its four adjacent neighbors. Each process runs on its own dedicated processor; it performs the iteration for a fixed, continuous subset of the grid array, which will be called a *task*. Thus, the processes are distinguishable. The algorithm was implemented on the 10-processor Cm* by Levy Raskin [55], and updated by Jarek Deminet [13] for the 50-processor system.

Several different variations [6] of this algorithm have been implemented on Cm*. All methods use one process per processor, so these terms can be used interchangeably. The processors iterate on equal-sized partitions of the grid.

1. *Jacobi's method.* At the beginning of each iteration, a processor retrieves its partition from a global array. New values are computed for each element of the partition, then stored back into the global array. This storing is performed inside a critical section. The processor then checks its error vector (computed from the difference of the new and old values in its partition). If the error vector is smaller than a pre-specified limit, the processor reports that it has finished. Otherwise it blocks, until the other processors have completed the current

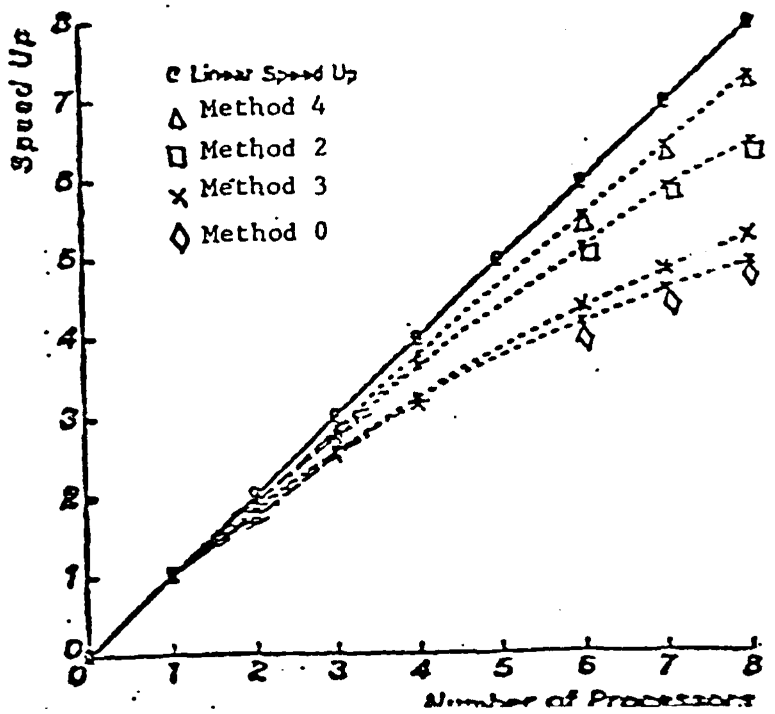


Figure 26: Comparison of Speedup for Different Methods of PDE

iteration. Iterations are performed until all processors have finished.

2. *Asynchronous Jacobi Method.* This method is the same as method 1, except that a processor does not wait for the other processors to finish before starting on the next iteration.

3. *Asynchronous Gauss-Seidel Method.* This method is similar to method 2, except that the processor uses newly computed values as soon as they are available, instead of the values known at the beginning of the iteration.

4. *Purely Asynchronous Method.* To compute new array values, this method uses the most recent values of all components by reading them directly from the global array and writing the updated values back to the global array (without any critical sections or synchronization). It uses a critical section only for a processor to report that it has finished.

Raskin compared these algorithms using a 21 × 24 array (504 elements) on a one-cluster Cm* system with a maximum of eight processors. His results are shown in Figure 26. With eight processors, method 1 yields a speedup of just over 5.0; method 3 gives a speedup of better than 5.3; method 2 yields 6.4; and method 4, almost 7.3.

An apparent anomaly in these figures is the lower speedup of method 2 compared to method 3, though the values used in asynchronous Gauss-Seidel iteration are nominally more up-to-date. However, the uniprocessor Gauss-Seidel algorithm is almost twice as efficient as the uniprocessor Jacobi algorithm. Because it cannot quite maintain this advantage as the number of asynchronous processes increases, the speedup of the asynchronous Jacobi algorithm is greater. In fact, all of the asynchronous algorithms take more iterations as the number of processors increases [6]; but the rate of increase with method 3 is greatest, followed by method 2 and then method 4. (The number of iterations taken by the synchronous Jacobi algorithm is independent of the number of processors.) Nonetheless, method 3 is still faster than method 2 for eight processors, by a margin of 38 percent. Beware of confusing speedup with speed; a slower algorithm may display a better speedup.

Note that the asynchronous Jacobi and Gauss-Seidel algorithms are the only two which demand precisely the same amount of synchronization (critical sections at the beginning and end of an iteration, no waiting for other processors). Regardless of execution time, each time the synchronization requirements are decreased—from method 1 to methods 2 and 3, and then again to method 4—speedup improves. Hence, the degree of synchronization seems to be the dominating factor affecting the speedup of the algorithm.

The cost of synchronization is also illustrated by a comparison of two methods of simulating the structure of liquids.

Simulation of Molecular Motion. Given the microscopic interactions between particles, we want to predict the static and dynamic properties of a collection of such particles. Macroscopic quantities are obtained by an averaging according to one of two methods—either ensemble averaging or time averaging. The Metropolis method [44] employs ensemble averaging. To generate each new configuration, a single particle is moved. The bottleneck of the calculation for each move is the computation of the binding energy for each particle, which involves $O(N)$ calculations. The parallel algorithm uses K processors in an attempt to reduce the complexity of this step to $O(N/K)$. The interactions can be evaluated in parallel without interprocessor communication, but the contributions calculated by each processor must be added together. The complexity of this step s $O(N)$. In addition, the computation must be synchronized at each move.

The molecular-dynamics algorithm [80] uses time averaging. First, an initial set of velocities for the particles is calculated. Given an initial set of coordinates, the velocities can be used to predict a set of coordinates at a later time. A summation is again performed to find the binding energy, but with this algorithm, the summation can be reordered to allow a processor to sum its subset of binding energies *locally*, with the global summation being required only once at the end of the computation.

These simulations are described by Ostlund, Hibbard, and Whiteside [49]. This problem is representative of the general problems involved in the theoretical study of molecular

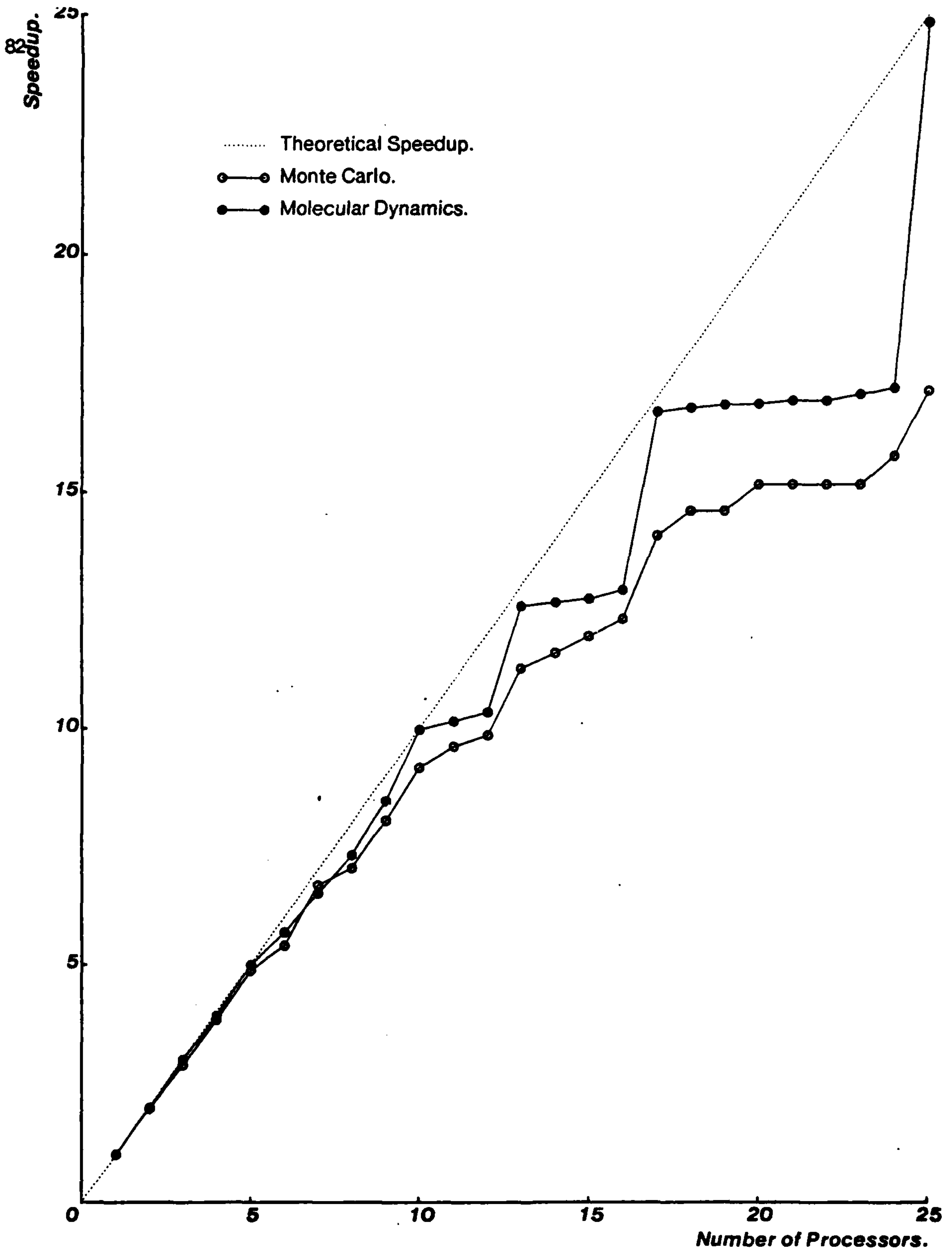


Figure 27: Speedup of Molecular-Motion Simulation

motion. The results shown in Figure 27 are for a system of 50 particles.

The molecular-dynamics algorithm shows better speedup than the Metropolis algorithm. One factor is that the molecular-dynamics algorithm avoids summing of shared variables. In addition, since particles move simultaneously rather than one at a time, the $O(N^2)$ serial computation of the binding energy is converted to an $O(N^2/K)$ parallel computation using K processors. A processor computes $O(N^2/K)$ interactions between synchronizations, instead of $O(N/K)$ interactions as in the Metropolis algorithm.

Both graphs exhibit a zig-zag pattern because the particles are parceled out among the processors as evenly as possible. Each time that there is a decrease in the number of particles handled by the "busiest" processor, speedup increases markedly. Note, for example, the large jump in going from 24 to 25 processors; in the latter case, no processor need handle more than two particles.

Both of these algorithms for molecular motion are synchronous, requiring lock-step iteration; no asynchronous molecular dynamics equation is known. Significant hurdles stand in the way of developing an asynchronous algorithm; for molecular dynamics, it would require averaging the interactions between particles occupying different positions in space *and time*. In general, even when an asynchronous algorithm exists, it is more difficult to prove its convergence. It must reliably attain the same final state each time it is repeated, yet it passes through a sequence of intermediate states which is unpredictable due to slight variations in processor speeds.

5.3.1.3. The Placement of Processes

On a multiprocessor like Cm* with a non-uniform reference time to different portions of memory, the placement of processes and data can have a major impact on algorithm performance. What factors need to be taken into account in deciding how they are to be distributed? The first is *locality of reference*. To expedite memory references, data should be local to the processor which references it. Although this is not feasible for shared global data structures, it still pays to minimize the distance between processors and their data.

For example, Raskin's version of the PDE assigned processors to array partitions essentially at random. But not all of the processors had the same amount of work to perform. Those that operate on partitions in the "middle" of the grid require more iterations than the ones that are close to the boundary, so ideally they should be solved by the fastest processes. When Deminet re-implemented the algorithm for the multicluster StarOS, he created a version that satisfied this criterion; we will call it the *improved processor selection* version.

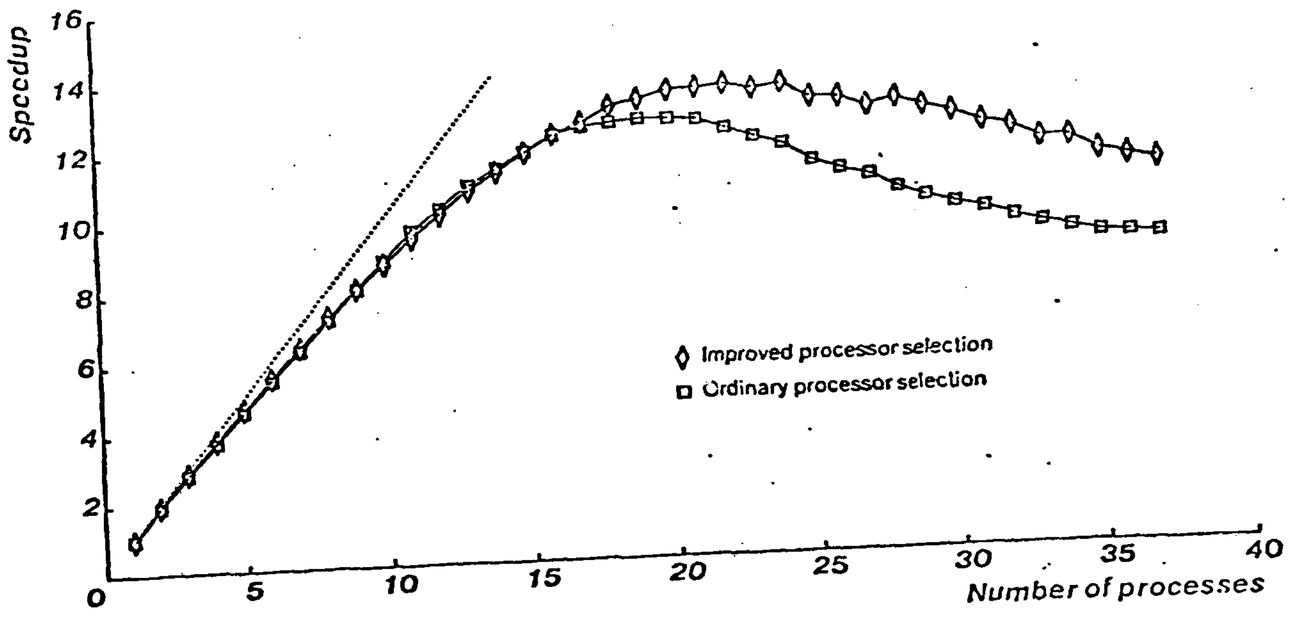


Figure 28: Effects of Improved Processor Selection

Figure 28 displays the effect of improved processor selection. It is significant when the number of processors is larger than 16, meaning that more than two clusters are involved. At thirty-five processors, the improvement is about 20%.

5.3.1.4. The Distribution of Data

While thoughtful placement of processes can greatly improve locality of reference, it does little to reduce *memory contention*. If a global data structure is deposited in the local memory of a single Cm, then many processors will compete for access to the same Slocal and memory, so response will deteriorate. The impact can be substantial because many multiprocessor algorithms, such as the PDE, make a large fraction of their memory references to a single shared data structure.

Deminet performed an experiment to determine the effects of data placement. Method 4 of the PDE was run on a 150-by-150 grid. In one case, the data was centralized in the memory of a single Cm. In the other, the pages of the array were distributed between clusters to maximize cluster locality of reference. Figure 29 displays the results. The centralized-data version achieved a maximum speedup of 16, while the distributed-data version yielded a speedup of 28 with 37 processors (the most processors for which it was run), apparently without reaching its maximum. It is also illustrative to consider the degradation of an

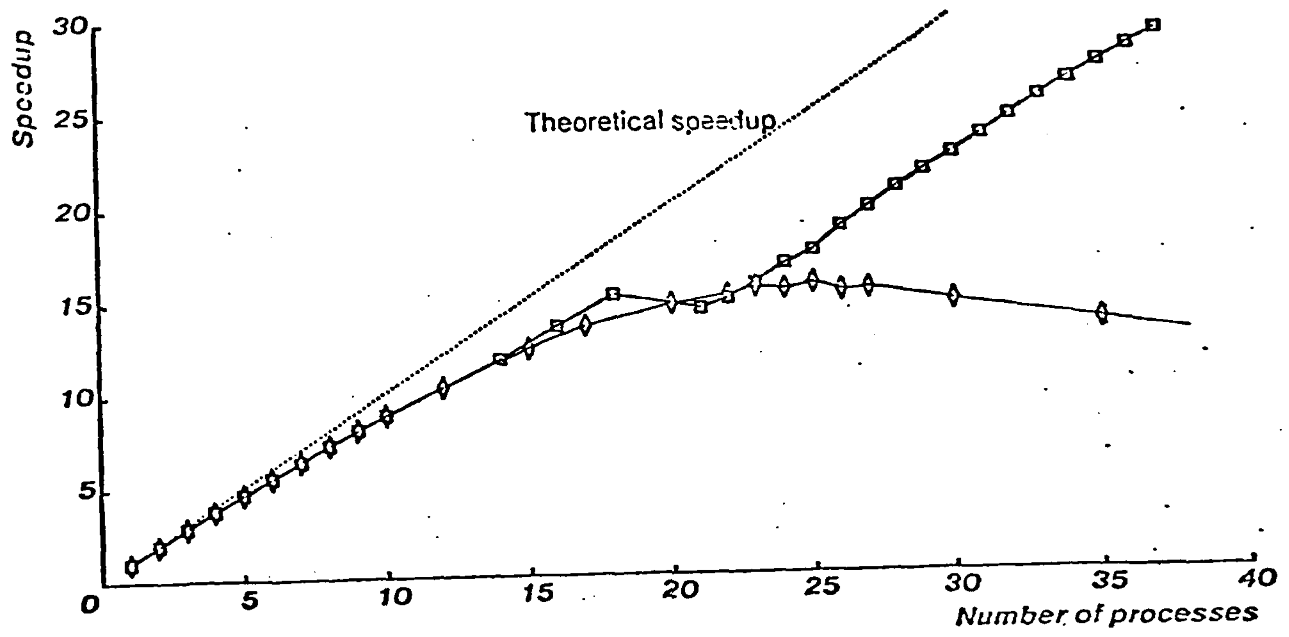


Figure 29: Effects of Distribution of Data

individual processor's performance: in the distributed-data version, all processors ran at least 68% as fast as the fastest processor, while in the centralized-data version, the slowest processor ran at only 28% of the speed of the fastest.

There is a tradeoff between locality of reference and memory contention. When data is too centralized, many processors are contending for the same memory. When data is too widely distributed, processors have to make too many expensive intercluster references. Consider again the PDE. When processors are added to the experiment, the strategy is to fill up one cluster before beginning the next. The curve in Figure 30 first shows a dropoff in speedup between 20 and 21 processors, just as the experiment moves into its third cluster. The explanation is that data is now distributed between three clusters instead of two. All but one of the processors need to perform intercluster accesses to access data in the third cluster. The extra intercluster references slow down the entire experiment. Notice the additional, and more pronounced, drops in performance as the experiment crosses into its fourth and fifth clusters. Deminet called this the *crossover phenomenon*.

As we have seen, many factors must be taken into account when deciding how to distribute data. There is no straightforward way to estimate the interplay of these factors and determine how to place the data objects in the most desirable location. Cm*'s operating systems provide no automatic distribution

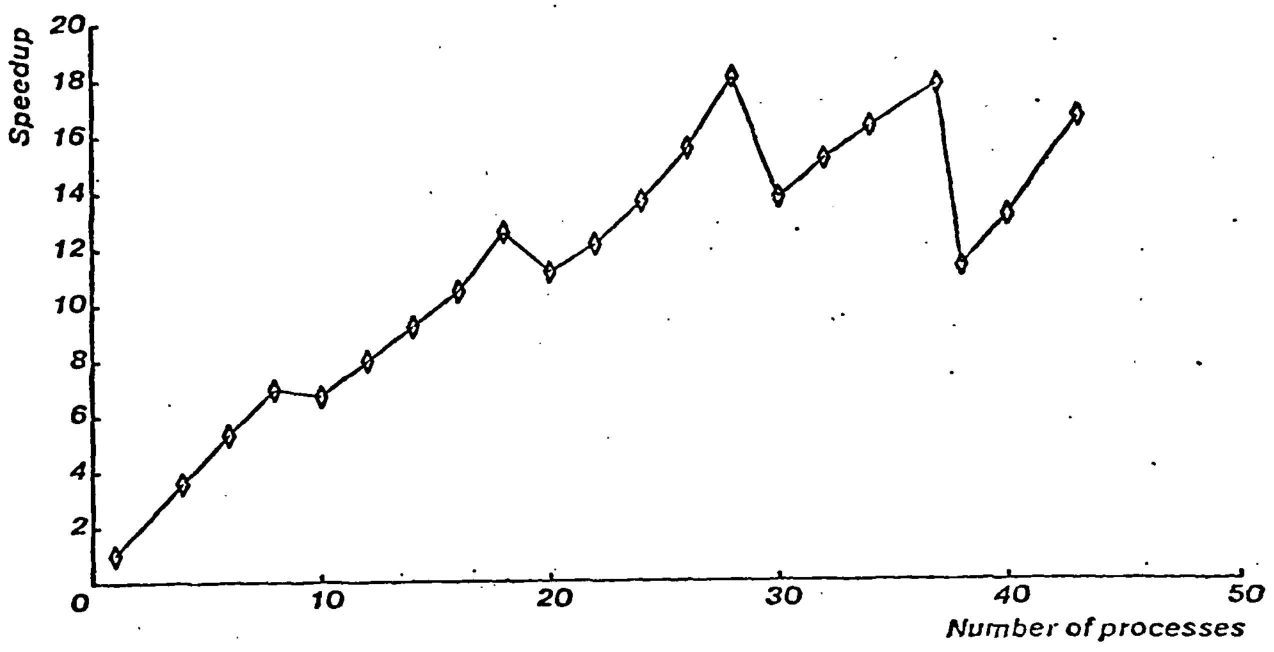


Figure 30: The Crossover Phenomenon with the PDE

algorithms. The TASK language permits the programmer to specify constraints on object placement, and uses heuristics to attempt to satisfy them, but it cannot guarantee optimality.

The Cm* quicksort is another algorithm which operates on a large shared data structure:

Quicksort. In the multiprocessor quicksort [60], a number of indistinguishable processes, one per processor, take part in sorting a global array of integers. The processors share a *stack*, which contains descriptors for continuous subsets of the array which have yet to be sorted.

On each pass, a processor tries to pop a descriptor for a new subset from the shared stack. If successful, the processor partitions the subset into two smaller ones, consisting respectively of all elements less than, and greater than, an estimated median value. After this partitioning, a descriptor for the shorter of the new subsets is pushed onto the stack, and the longer subset is further partitioned in the same way. This algorithm was implemented on the 10-processor Cm* by Levy Raskin [55]. Jarek Deminet [13] adapted it to the 50-processor configuration, modifying the algorithm to cut down on references to the shared stack.

Placement decisions are simpler to make if an algorithm has a predictable pattern of reference, for

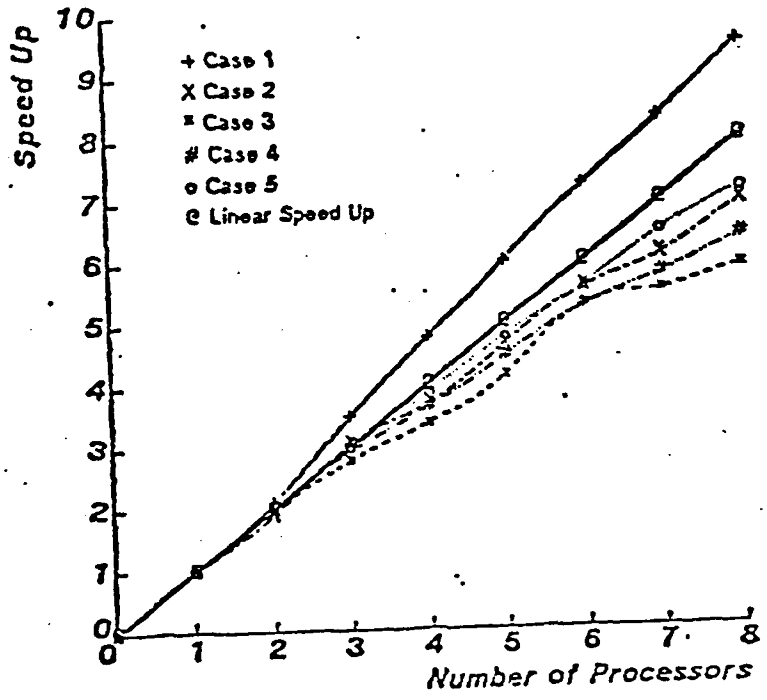


Figure 31: Speedup of Quicksort

example, if each process is statically associated with a partition of data which is known at compile or load time. The PDE is such an algorithm; hence it is possible to arrange for the data to be close to the processor, and it displays an increasing speedup even for relatively large numbers of processors. On the other hand, the quicksort's processors dynamically choose subsets of the global array from a shared stack; it is not known *a priori* which processor will manipulate which data. The quicksort (see Figure 31) reaches maximum performance at a relatively small number of processors.

5.3.1.5. Greater Than Linear Speedup?

Is it ever possible for a multiprocessor algorithm to exhibit greater than linear speedup? Intuitively, the answer is no, because such an implementation would do less total work than its uniprocessor cousin. By simply turning its independent processes into coroutines, one could produce a faster uniprocessor version of the same algorithm. This assumes, of course, that as much memory was available to the uniprocessor version as to the total multiprocessor version; otherwise more frequent swapping would retard its execution speed. Also, if process switching were much slower than synchronization primitives, the multiprocessor version might gain a more-than-linear advantage over the coroutine version, but this could not properly be called speedup.

Nonetheless, we do occasionally encounter a particular algorithm *execution* with more-than-linear

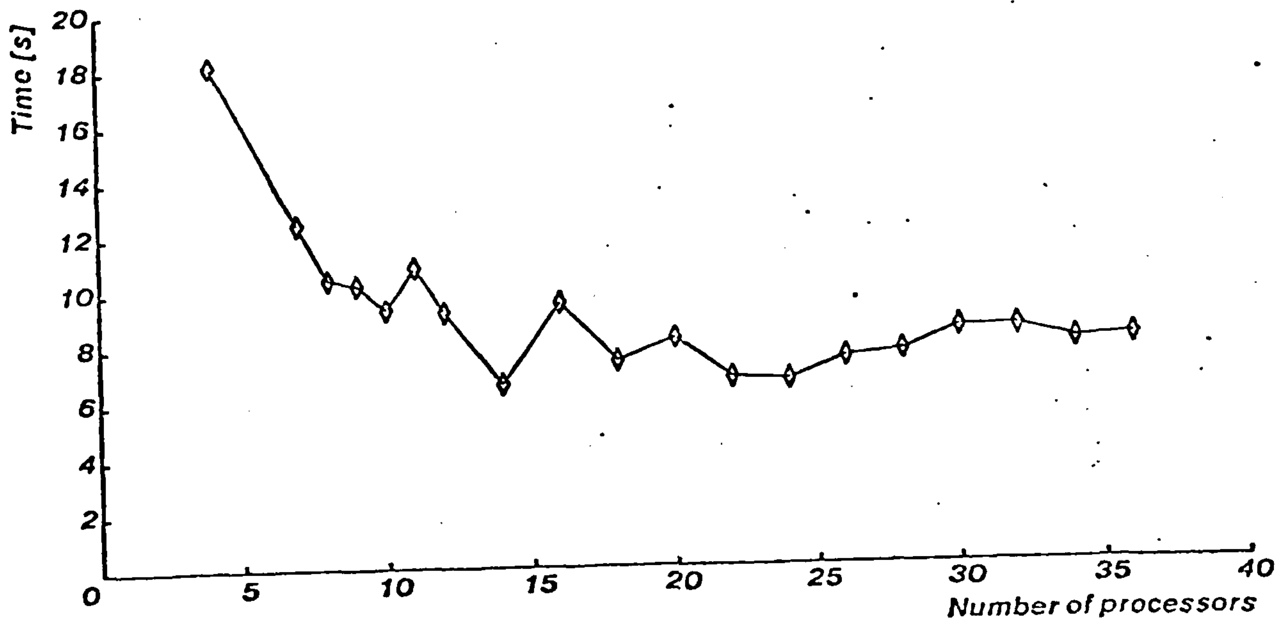


Figure 32: Speedup of Integer-Programming Computation

speedup. One such case occurred with the integer-programming algorithm first mentioned in Section 5.2.2, which was implemented by Raskin for the 10-processor Cm*. This is a search algorithm. Its initialization phase puts a large number of possible solutions in a global stack, from which all the processors choose their work. As the search proceeds, a global variable holds the cost of the best solution found so far by any processor. All processors compare their current cost value to it, and begin to backtrack in the search when the global cost is lower.

It is possible for the multiprocessor version to be "lucky". If one of its processors encounters a near-optimal solution at the outset, none of the processors will have very much work to do. The uniprocessor version, which does not encounter the near-optimal solution until later, has the disadvantage of having done a more complete search over the earlier possible solutions. But the opposite can also happen. Suppose the near-optimal solution turned up, say, first of all. Then the uniprocessor and multiprocessor versions would encounter it at the same time. But before its cost could be determined, the other processors in the multiprocessor version would have wasted processing time on their initial solutions.

Hence, the multiprocessor version cannot be "lucky" all the time. A search program can occasionally exhibit greater than linear speedup, but it cannot "on average" show greater than linear speedup, over all possible sets of input data. The results shown in Figure 32 illustrate this. Only one of the five integer-programming ru [illegible]nanaged to surpass linear speedup.

5.3.2. Task-Force Organization

Implementations of the algorithms introduced in the previous section are quite simple. A single *master* process is responsible for setting up and starting each of the other processes, known as *slaves*. The slaves execute in parallel, perhaps synchronizing from time to time, until they have completed their portion of the computation, and then notify the master. When all of the slaves have completed, the master reports the results of the experiment.

Master-slave organization is one of the simplest task-force structures, and one for which it is meaningful to measure speedup, since the multiprocessor algorithms have a uniprocessor analogue which can be used for speed comparisons. Some algorithms have no uniprocessor version, for example, this simulation:

> *Railway-Network Simulation.* The task force consists of a fixed number of processes (63 in this version), each of which represents a *station*. Two stations may be connected by a unidirectional *track*. For a given station *A* a set of *previous stations* includes each station *B* for which there is a track from *B* to *A*. The stations exchange messages representing *trains*. The route of each train is an attribute of the train, determined when the train is created. Each station serves the trains in the order in which they arrive.
>
> Each process maintains its own simulated time. At any given moment, the simulated time will probably be different in different processes; thus the simulated time of sending a train from one station to another is unrelated to the real time when the message representing the train was created. Even the real-time order of events may be different from the simulated-time order. Suppose, for example, that station *A* sends a message to station *C* at real time 5. At this moment station *A*'s clock may show the simulated time 50. Station *B* may send a message to *C* at real time 7, but its clock may then show the simulated time 40. The second message should be serviced by *C* before the first one, since only the simulated time is relevant. In general, a station-process blocks until it knows the simulated time of the arrival of the next train from each station.
>
> Unlike in the previous experiments, several processes may run on each processor. There exists one additional process, the *reporter*, which records data sent to it by the stations. As programmed, the application may not be run on a uniprocessor configuration,

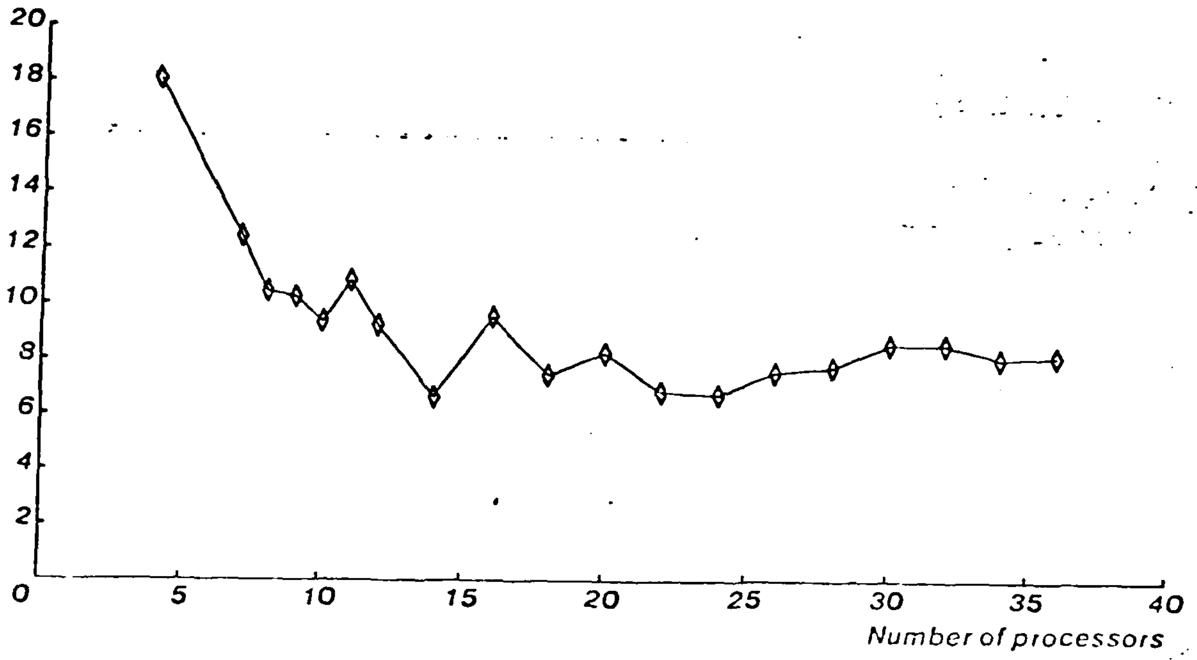


Figure 33: Speedup of Railway-Network Simulation

since the reporter must be running throughout the experiment. In fact, this task force will fit in a minimum of four Cm's. The algorithm was programmed by Jarek Deminet in 1979.

It is very difficult to estimate speedup for this algorithm, because running time depends not only on the number of processors, but also on the distribution of processes among them. In the implemented version, processes may not move from one processor to another. If they did, it would probably be necessary to move their local data to avoid the penalty of remote references. To move data would itself impose an overhead. Thus the run time of the experiment depends heavily on the initial assignment of processes to processors. Figure 33 graphs the run time vs. number of processors. After an initially decreasing region where the average number of runnable processes is greater than the number of processors, there are apparently random fluctuations in the graph, perhaps due to variations in the suitability of the initial assignment of processes to processors.

5.3.3. Hardware Reliability Tests

Long-term reliability data for the components of Cm* has been obtained by the Auto-Diagnostic, a program which continuously exercised the hardware by running diagnostic programs on all otherwise idle computer modules. The diagnostics tested four distinct components: the memory, the instruction set, the traps and interrupts, and the Slocal and a small part of the Kmap. The memory test consists of thirteen subtests including a gallop test, marching ones and zeros, and shifting ones. The instruction-set test and the trap-and-interrupt test check the functioning of the LSI-11 processor. The Slocal diagnostic tests the registers and data path of the Slocal, among other things.

Table 1: Distribution of Transient Errors on Cm*

	Diagnostic Test Used					
Mode	**Memory**	**Instr.**	**Trap**	**Slocal**	**Total**	**% Total**
Burst	5	1	1	13	20	31.2
Simultaneous	5	1	0	5	11	17.2
Isolated	13	3	2	15	33	51.6
Total	23	5	3	34	64	100.0
% Total	35.9	7.8	4.7	51.6		

In continuous measurements over a period of eight months, transient errors were noted more than twenty times as often as hard failures (permanent faults which reflect an irreversible change in the hardware. Of the transient errors, 83% of them were local, confined to only one Cm. Table 1 [78] reports the data. A *burst error* consists of multiple errors in the same Cm within a short span of time. *Simultaneous errors* affect more than one Cm; an *isolated error* is a single error in a single Cm. Transient errors followed a decreasing failure-rate Weibull distribution. This is in contrast to the usual assumption of exponentially distributed transient errors. Studies done on several other computers also reported a decreasing failure rate [42].

5.4. Summary

We have tried to make general observations about the cost effectiveness of multiprocessor computation, based on a variety of experiments on Cm*. These are only a sample of the work that has been done with Cm*. Aided by the experimental support provided by the Cm* test bed, a flexible hardware/software architecture, we are continuing to broaden the range of our investigations. The long-range issue is not whether Cm* itself, with its relatively primitive microprocessors, is a useful vehicle for solving large problems, but whether multiprocessor architectures in general have an important role to play in providing computational power.

Multiprocessors have some important advantages over uniprocessors. Since simple processors are becoming quite inexpensive, multiprocessors of substantial size may prove significantly less expensive than uniprocessors of the same power. Their total processing power is not so constrained by fundamental physical constants as is the case for uniprocessors. Their cost advantage leads to another benefit: they can be cost effective without very high processor utilizations; they can afford to lose some computational cycles to synchronization, for example.

We are optimistic that the benefits of multiprocessing can be realized in practice. Many algorithms have a theoretically linear speedup, including the asynchronous partial-differential equation solver and the molecular dynamics algorithm presented in this paper. Other algorithms, such as the quicksort and the power-systems simulation, do not decompose so effectively. There may be some *problems* for which there is no good multiprocessor algorithm; for example, those that converge only with a high degree of synchronization, or those that have memory-access patterns so unpredictable that data cannot really be localized. In between these extremes, much more work is necessary to refine and automate the techniques for decomposing algorithms and distributing processes and data. Our approaches have often been heuristic, but they can provide the basis for more extensive and rigorous investigation.

AD-A120 056   CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
REAL TIME STATUS MONITORING FOR DISTRIBUTED SYSTEMS.(U)
AUG 82 Z SEGALL, A.SINGH, R SNODGRASS   DASG60-80-C-0057

UNCLASSIFIED   NL

2 OF 2
AD A
120056

END
DATE
FILMED
11-82
DTIC

Acknowledgments

Cm* is a large project. The authors wrote the paper, but performed only a fraction of the work. Even this list of contributors is of necessity very incomplete. Richard Swan, Sam Fuller, and Dan Siewiorek were major instigators of the hardware design. John Ousterhout was involved in the software effort from the start, and along with Pradeep Sindhu, was primarily responsible for the MEDUSA operating system. Anita Jones, Bob Chansler, and Ivor Durham, along with Karsten Schwans, Steve Vegdahl, Mike Kazar and Joe Mohan, designed most of the STAROS system. Zary Segall, Rick Snodgrass, and Ajay Singh were the driving forces behind the IIE. Levy Raskin was the first experimenter, and he was followed by Jarek Deminet, Neil Ostlund, and Bob Whiteside. Sarosh Talukdar oversaw the power-systems simulation work.

The Status Monitor

6. The Problem

R. Snodgrass

The cause is hidden, but the result is known.

-- *Ovid, from* Metamorphoses IV

6.1. The Cause and the Result

In the realm of programming computers, as in all analytic endeavors, one must first understand the behavior before one can understand the underlying reasons for that behavior. As the computational structures employed in programs tend toward greater complexity, computer system designers, implementors, and users find it increasingly rare that they can agree with Ovid that "the cause is hidden, but [at least] the result in known." Monitoring is a necessary first step in understanding a computational process, for it provides an indication of *what* happened, thus serving as a prelude to ascertaining *why* it happened.

The realization that monitoring is a difficult task, one that deserves study, has come only recently to the computing community. The reason for this slow awakening is, again, the increase in complexity. When computing systems (both hardware and software) were simpler, it was possible to adequately understand the system's behavior with rather unsophisticated monitoring tools and the (considerably more sophisticated) modeling techniques then available. Many aspects, such as characterizing the control flow or determining execution times, were so straightforward as to not even being considered issues. Times have changed, and many of these "non-issues" are now so problematic that systems often do not provide *any* solutions to them.

This chapter (and the thesis in general) is loosely organized around a sequence of problem and result statements. Implicit in each problem statement are the results generated by preceeding statements, since one benefit of acquired knowledge is the ability to ask further, more precise questions. Each result will bring about other questions, which will impact on the ensuing problem statement. At this point, the problem statement is annoyingly vague:

> **Problem 1:** What is involved in monitoring distributed systems, and why is it such a hard problem?

The purpose of this chapter is to refine this problem statement into one which is more concrete and thus more approachable.

6.2. Definitions

The definition of monitoring employed in this dissertation is a rather general one: monitoring in the process of extracting dynamic information concerning a computational process as that process executes[2]

.

A *computational process* is simply anything that can be said to compute. Examples include the microprogram, a subroutine, a conventional process, a collection of processes, or even an operating system.

Computational processes differ primarily in the number of components to be monitored, from a single line on a bus to the entire system, and in the time frame in which the measurements take place, ranging from hundreds of nanoseconds to months. The granularity at which the monitoring takes place has a substantial effect on the methods used to collect data: different granularities demand radically different approaches.

Dynamic information may also be spread over a large range of granularity, from information concerning the sequence of micorinstructions executed during a particular time interval, to the average amount of time a routine executes, to some global statistic concerning the execution of a whole series of programs. This property goes hand in hand with the last part of the definition. If the information to be collected is not dynamic, there is no need to collect it as the process executes.

Defining a distributed system is more difficult. Although John Shoch has several arguments to support the contention that "there is nothing different about 'distributed' computing" [62], he also presents several distinctions between distributed and non-distributed systems (his widely-shared belief is that there *is* a difference). The two relevant to monitoring are

- Distributed systems seem to be characterized by a lack of central control.
- A quantitative difference in the number of system components (processors, memory, addressing domains, etc.) leads to a qualitative difference[3]. This will not be possible for distributed systems. How can we comprehend parts of the system without comprehending all

[2]There are at least two other definitions of *monitor* which should be mentioned. One use of the word monitor, prevalent in the 1960's and early 1970's, is as a synonym for *operating system* or at least the user interface to the operating system. The second refers to an arbiter of access to a data structure in order to ensure specified invarients, usually relating to synchronization [25]. Both definitions emphasize the *control*, rather than the *observational*, aspects of monitoring.

[3]At the same workshop [38], Richard Watson added several related attributes: more heterogeneity, distribution of state, and communication via messages. And David Reed offered perhaps the best argument for monitoring in his characterization of distributed systems: "*In centralized systems, it has been possible so far for single persons to understand the entire system (even of the size of* MULTICS

of it?).

These two aspects conspire to make monitoring a difficult (and thus interesting) task. A precise definition of 'distributed' is not important; the intent of the title is to include the attributes listed above in the problem domain.

The general definitions presented above allow concepts developed in this research to be applied to several previously unrelated domains. This section closes with a discussion of representative utilizations of monitoring information.

One use of monitoring is to facilitate the debugging of complex programs. Debugging proceeds in five stages: (1) observe the behavior of a computer program; (2) compare this behavior with the desired behavior; (3) analyze the differences; (4) devise changes to the program to make its behavior conform more closely to the desired behavior; and (5) alter the program in accordance with these changes [45]. Monitoring is concerned with the first (and somewhat the second) stage in this process. The third and fourth stages are still the province of the programmer (although the Programmmer's Apprentice project [64] is making some progress in this area); the fifth stage is routinely accomplished using text editors, and could be automated given the automation of the fourth step.

Sophisticated monitoring tools are necessary to make efficient use of limited computing resources. Ideally, such optimization would be done analytically, but in general a priori determination of runtime efficiency is impossible. Thus it is necessary to tune the application once it is implemented. Tuning requires feedback on the program's efficiency, which is determined from measurements on the application while it is running[4].

A third use of monitoring is to query the system, not for performance measures, but merely for status information, such as how far a computation has progressed, who is logged on the system (the *system status* command of most time-sharing systems), the state of certain files (the *catalogue* or *directory* commands), or the quantity and nature of hardware and software failures.

And finally, monitoring information may also be used internally by the application program for various purposes. For example, consider a program which varies the number of processes dedicated to a particular function based on the request rate for that function. Information concerning the hardware

[4]Robert Sproull called this tuning *performance debugging*: it's not enough just to show that a system works; you want it to work well [38]

utilization and the number of outstanding requests could be used by the program to determine whether to start up more processes to handle the current demand (if the utilization is low and the request rate high) [52]. Monitoring information is also valuable for programs which must be reliable; the fact that a processor (containing particular processes belonging to a program) has failed, for example, is important to the program if it must be able to recover from such failures[5].

6.3. The Impact of Complexity on Monitoring

The previous section indicted that monitoring is difficult because of the complexity and decentralization of the process being monitored. The purpose of this section is to determine how increased complexity impinges on the task of monitoring. The impact of decentralization is reflected more in the specific algorithms and will be dealt with in later chapters. We will start by investigating the monitoring of the program counter, or PC, certainly an important aspect of the dynamic state.

In the (good old) days, a program consisted of a single program executing on a monolithic operating system on a single processor. The PC could be traced or sampled. Tracing, which involves storing the information each time some event occurs, is usually done at the procedure level, although it can be done at the statement level with greater overhead. At the beginning of each procedure (or statement), code is inserted by a preprocessor to increment a counter or generate a timestamp. A postprocessor is often used to correlate the data with the source text of the program.

Sampling involves storing information when requested, asynchronously with the execution of the program. Usually sampling is initiated by a clock tick, by an operating system call, or by a separate process. The information gathered by sampling is stochastic; for instance, it can indicate what percentage of execution time takes place within an individual routine, but it cannot reliably determine how many times a routine was invoked. Sampling does have the advantage that it requries less resources, and thus perturbs the system to a smaller degree than tracing.

In the past two decades the programming environment has changed radically. In some sophisticated systems being developed today, a program consists of many interacting processes running on many geographically distributed computers communicating over high bandwidth networks [Xerox]. These systems differ quantitatively with systems of the past: where there was one processor, there are now tens to hundreds; where there was one process, there are now many per processor; where there were a few

[5]Eric Rosen, in an article describing a particularly interesting instability which occurred on the ARPANET [57], concluded that "we need a better means of detecting that some high priority process in the Imp [a node on the ARPANET], despite all the safeguards we put in, is still consuming too many resources."

I/O devices, there are now complex communication media, sopphisticated encoding formats, and powerful interprocess communication protocols, all supported by large software components; where there was a single contiguous address space, there are now many small, separately addressable objects, each containing code or a specific data structure.

Returning to the example of monitoring the PC, we must first determine what the PC is for this new environment. One possibility is to use the PC of each of the processes making up the program of interest. For the single process example, the routine name and statement number within that routine can be quite informative; a printout of, say, fifty routine names and statement numbers is rather overwhelming. This *quantitative* difference necessitates a *qualitative* change in the monitor, for there is one aspect that remains unchanged: the user (and especially the information capacity of the user) who must interpret the monitoring data.

The presence of the user has been implicit throughout this discussion. Fundamentally, the user is not interested in the PC at all; instead, the user wants an understanding of the *state* of the execution as it evolves through time. This state manifests itself in many forms: the changing values of the variables in the program, the input read by the program and the output produced, the constantly changing PC. All are valid indicators of the program state and each may be sufficient when monitoring a single process. Individually, however, they are woefully inadequate for monitoring distributed systems. Instead, the monitor must be able to express the system state (as well as other attributes of the system) in a form useful to the user.

As an example, suppose the monitor could provide this description of the program state:

> Process A is waiting on process B to acknowledge the xxx request; process Y is sending process Z information concerning the object yyy; and process M has completed.

There are several aspects to note in this example. The information the monitor displayed is both less and more than a list of PC's. The monitor had to understand that a PC in a certain range meant that process A was waiting for something, yet the exact PC was unimportant. Conceivably, the PC could have been completely different and the monitor would have displayed the same information. In addition, the monitor had to be able to look inside the various queues and buffers maintained by the communication mechanism in order to be able to state that a process is waiting on another process to acknowledge a particular request. Names had to be associated with the various processes, objects, and requests in order to produce an intelligible state description. And finally, the monitor had to know that the user was interested in the current state in terms of interprocess communication. Another perhaps just as useful state description is

Process A has used 75% of its resources, while processes X, Y and Z have used only 20% of their resources.

The decentralization inherent in distributed systems also necessitates *interpretation of the monitoring data* by the monitor. The mention of several processes in the previous example implies a degree of logical decentralization; if those processes are on different processors, then there is also physical decentralization. To present a global view of the program state, the monitor must integrate data collected at geographically distinct sites. Simply determining what information to collect and where to acquire this information becomes a difficult task.

Result 1: The quantitative and decentralized aspects of the monitored system, coupled with the limitied information handling capabilities of the user, demand an intelligent monitor.

At this point, the problem at hand is

Problem 2: What are the organizing concepts for a monitor which can collect information from a variety of sources, interpret this information, and present it in a series of high level views in a format comprehensible to the user?

6.4. Knowledge Representation

In its most general form, the process of monitoring is concerned with retrieving information from the monitored system and presenting this information in a derived form to the user. Viewing the monitor as the proverbial black box, it is fundamentally an information processing agent. As the previous sections have indicated, this activity is rather sophisticated. Looking inside this black box, there is some form of knowledge representation to direct the monitoring activity. Thus, there are at least two ways to abstractly view a monitor: as a *knowledge representation system* and as an *information processing agent*. As will be seen, both of these views are very fruitful. The rest of this chapter will investigate the knowledge representation issues; chapter 2 will pursue the information processing aspects of monitoring.

In an examination of the discussion of a possible high-level PC, one starts to notice phrases such as "the monitor had to understand." Now, in one sense, the monitor *can't* understand; it is, after all, only a computer program. However, computer programs are remarkably versatile and almost any type of desirable behavior can be programmed with the correct selection of data structures and algorithms. Hence, the process of "teaching the monitor" or "making the monitor understand" is transformed into the more intellectually managable task of deciding what data structures and algorithms to employ within the monitor.

These data structures and algorithms encode the knowledge the monitor can apply to the task at hand. The majority of monitors did very little interpretation of the collected data, and thus used rather ad hoc methods for determining what to monitor and how to perform the monitoring. Two recent systems have addressed the monitoring of complex systems; it is useful to analyze the character of knowledge each used to direct the data collection and interpretation.

Model's thesis [45], one of the first to systematically approach this topic, stressed the adoption of a uniform model of a complex activity for use in monitoring. His monitor was designed to be used with programs written in artificial intelligence languages such as KRL, which are themselves written in Lisp. Despite the sophisticated control and data structures provided by these high level languages, most debugging is still done in the implementation language. The complexity of programs written in these languages is seriously limited by the lack of adequate debugging tools. Model argued that of the five stages present in the debugging process (see section 6.2), monitoring has the most potential for improvement at this time.

The monitor collected events generated by the interpreter (the monitor had no control over which events were collected). These events were related to the program's data and control structures implicitly in the routines generating the events. However, some cross-referencing was done, so that the monitor knew, for example, that some events caused other events. The user could specify which events, as well as which fields in these events, were to be displayed. The knowledge utilized by the monitor was wired into the code.

Gertner's thesis [22] focussed on the flow of messages between processes in RIG [4], a distributed system constructed at the University of Rochester. In his system, Gertner described the computation using finite state automata, with the transitions being events (usually messages sent between two processes in the system). Associated with each message is a set of timestamps relating to the activity involved in processing the message. These timestamps allow the monitor to calculate processing intervals, message counts, overlapping periods, etc. A hierarchy of finite state automata can be defined, with elementary transitions at one level composed of multiple transitions at a lower level. This hierarchy allows monitoring information to be presented at the appropriate level of abstraction. Again, the knowledge of how to derive information from the timestamps was implicit in the monitor's code.

Unfortunately, these ad hoc approaches are simply inadequate for distributed systems. In Model's system, events capture only the notion of *state transitions*. The system state must be inferred by the user. Modelling all activity in terms of finite state automata, as in Gertner's system, while expressing to some degree the semantics of the periods between the events, is overly restrictive. Sampling data (as opposed to trace data) does not integrate easily into the scheme. The proliferation of extraneous states is also a problem which results from a total reliance on this model.

In order to construct a monitor which can apply substantial knowledge concerning the system being monitored, this knowledge must be organized in a coherent fashion. Thus a formalism is needed to describe this knowledge. The formalism must, to some degree, encode the following knowledge:

- what information the monitor collects concerning the system;
- how new information can be acquired by the system;
- what dependencies exist between various components of this information;
- how the information relates to the data and control structures within the programs, and to the data and control structures of the underlying operating system; and
- what information the user wants to see.

There are also three basic "notions" that must be characterized by this formalism. Two of these are the concepts of "entity" (or "object") and the concept of "relationship" between two or more entities. The monitor must understand that there are such things as processors, processes, memory, message ports, semaphores, etc. and that certain relationships exist between these things, such as a process running on a processor. In Model's thesis, for example, entities were the values of certain attributes, and the relationships were the events themselves.

The third notion that must be characterized by this formalism is the concept of time. The monitor must understand that facts are only true for a certain period of time, and that entities and relationships are temporally bounded. For instance, in Model's thesis, time was one of the fields in each event record, and queries could specify which time period the user was interested in. Also, Model's monitor understood that events were sequential, and thus that some events were after others. However, the concept of something being true for a period of time *between* two events is not represented within the monitor, and thus the user could not request such information. Clearly, a multiprocess monitor must have a better understanding of time.

The aim of this chapter has been to sufficiently refine the original problem statement into one which can be attacked in concrete terms. This chapter has argued that

> **Result 2:** Monitoring in fundamental terms is concerned with knowledge representation and information processing.

This leads to the following problem statement:

> **Problem 3:** What knowledge is necessary to adequately monitor a distributed system, and how is this knowledge represented within the monitor?

However, before we can address this question, we must investigate the information processing aspects of monitoring.

I shall limit the scope of my investigation to the above problem. I will not deal in depth with the uniprocess aspects of monitoring; instead I will concentrate on monitoring process-process and process-operating system interactions (the distinction between these interactions is lessening in importance [53]).

7. A Low Level Data Collection Mechanism

R. Snodgrass

Data collection techniques have been at the center of attention in previous work in monitoring, to the exclusion of other areas such as representation and manipulation of the collected information. Most papers on the monitoring of user programs describe *profiling* in a variety of programming languages. This approach involves execution counts or timing at the procedure or statement level, using sampling or tracing. These papers are simply variations on a common theme; there have been few advances since the early 1960's, when sampling and tracing were first introduced. Data collection for monitoring of operating systems has also relied on sampling or tracing Techniques for using special hardware have been more innovative: since additional logic imposes no overhead on the computation, capabilities such as event counters, combinational and sequential logic on events, comparators, and histogram generators can be provided. Network data collection has concentrated on performance evaluation issues and has been ,in general, confined to techniques mentioned above. Recent systems have taken a more integrated approach to monitoring, attempting to reduce the great effort necessary when using the low-level tools previously available. A unified set of facilities for monitoring a packet radio network was developed at UCLA. Gertner's thesis, described earlier in section, allowed relatively painless monitoring of a distributed system at the message passing level. The Computer Network Monitoring System (CNMS), a rather ambitious system designed at the University of Waterloo, used a sophisticated combination of hardware and software to monitor a geographically distributed network.

In spite of these developments, an integrated approach to monitoring data collection founded on fundamental concepts, goals, and limitations is still lacking. One possible strategy for developing such an approach is to start with the relational model described in the previous chapter. Unfortunately, this strategy is insufficient. Data collection must be strongly tied to the primitive relations referenced by the user. The operationalization of a primitive relation, is a predicate which, when presented with a possible tuple, will query the system and determine whether the relation is satisfied for that tuple. This predicate provides a well-defined semantics for the relation, but does not allow a direct implementation.

Another possible strategy proceeds by developing a conceptual model of the behavior of the program to be monitored, and attempts to represent that behavior witin the relational model. This strategy will be the one pursued in this chapter. The next section begins with a comparison of data collection as performed by a conventional data base system and by a monitor. A model of the environment where the data collection takes place is then presented, followed by a discussion of the properties an effective mechanism must have, The remainder of this chapter will present such a mechanism and examine how various aspects of the environment impact on this mechanism. The discussion will be independent of any particular operating system However, it is assumed that the monitor is partitioned into a *resident* portion, responsible for collecting the event records and interacting with the operating system, and a system-independent *remote* portion, responsible for analyzing and displaying the monitoring data. This separation is necessary when monitoring a distributed system, where a resident monitor would exist at each processor, sending collected data to the centralized remote monitor. The implementation follows this organization.

7.1. The Environment

Data collection in the monitoring domain differs from that in conventional database management systems in several ways. In most information processing systems, the emphasis is on information manipulation and retrieval, with mimimal aids for data collection. Although some systems provide tools for key entry and point-of-sale data acquisition, data collection remains difficult to automate. The reason is that the highly-structured databases must interface with much less regimented mechanisms: written and oral communication, multiple incompatible data representations, psychological and societal constraints. The monitor, as an information processing system, has much more control over the collection of data, since that data is already available in digitized form, either resident on a bus line or network link, or stored in registers, main memory, or on disk, certainly in a most convenient format.

The availability of monitoring information results in a second distinction between data collection as performed by conventional information processing systems and by the monitor: rather than being constrained by insufficient data, the monitor must take dramatic measures to *reduce* the incoming flow of data. For example, suppose that the monitor receives a value-time pair for each change in the program counter. The monitor would have to run on a machine several orders of magnitude faster than the one being monitored merely to store the information. However, if only routine timings were desired, the grain of data should be much coarser. As another example, suppose that timings were desired only for a single routine. Unless the data collection mechanism supports *filtering*,where only data satisfying specified constraints is actually collected, the monitor will have to contend with data concerning all routines. Extraneous data is very expensive to collect, since computing resources are required to collect it <u>and</u> to decide to discard it. Thus, data collection for monitoring involves careful selection of data, rather than access and conversion to a more useful representation, as in conventional information processing systems.

In order to discuss the data collection mechanisms within the operating system or application program, it is necessary to characterize the environment in which the mechanism executes. The environment is defined to be a collection of *strongly typed objects*, both passive (e.g. data structures) and active (e.g.

processes). *Type managers* export functions to be applied to objects of the type(s) supported by the manager; all operations on an object are performed by the tyr◦ manager supporting that type as requested through well-defined interfaces (implying the existence of a type-checking mechanism). This model thus identifies the *operation* being performed on the *object* by the *performer* (the type manager) as a result of a request by an *initiator*. The user can create new types by defining the representation of the object and specifying the operations which can be performed on objects of that type. The model applies to all levels of granularity; in particular, a type manager may be implemented in hardware, firmware, or software.

The model employed here has been used in several recent operating systems [27] and languages, although the model can be used to conceptualize program behavior in any system). The model is especially applicable to monitoring because it forces data structure state changes to be precisely specified: any change to the representation of a data structure (i.e., object) must occur within a function of the type manager as a result of performing a defined operation. Control flow can also be characterized in this manner: all changes in the execution state of a process can be accounted for by examining the sequence of operations performed by the process. Monitoring the model directly is difficult, due to the large differences in granularity involved. For instance, monitoring the add operation performed on an integer object by the hardware concerns significantly different issues than monitoring the file read operation performed on a file object by the file system.

There are several properties related to the typing model which should be satisfied by the data collection mechanism. The mechanism should support strong typing, in that typing violations are not necessary to perform the data collection. The mechanism should rely as little as possible on cooperation by the type managers, to allow additional data types and type managers to be monitored easily. The mechanism should be efficient, especially when disabled. The mechanism should be very flexible, allowing the monitor to specify exactly the information to be collected, thereby supporting filtering. And finally, the mechanism must exhibit good software engineering.

One property not listed above is simplicity. Often the attributes of efficiency and flexibility interact with simplicity; for instance, a mechanism supporting many modes of operation may be flexible but will probably not be simple. Also, while restructuring an algorithm can make it both less complex and more efficient, squeezing out the last bit of efficiency usually leads to more complex code.

Given that there are trade-offs, our approach is to place simplicity below the other properties in importance. This strategy is acceptable given an intelligent monitor which can take a high-level, non-procedural query and map it into a series of requests to the low level collection mechanism. Thus, simplicity in the low level mechanism has only a second order effect on the complexity of the user interface. As was argued in chapter 1, monitoring distributed systems *is* complex; this complexity will be evident throughout the monitor, except at the level of the user interface. We assume that the higher levels of the monitor can contend effectively with the complexity (and flexibility) presented by the data collection mechanism.

7.2. The Mechanism

The data collection mechanism employed in the monitor is closely tied to the type model presented earlier. An *event* is simply an occurrence the user is interested in. Each event occurs in the context of an operation as defined in the type model. Information concerning an event is structured into an *event record*, which is potentially of variable length and which contains both system and user-defined fields. Event records are typed, with each event type being produced by a particular *sensor*, which determines that an event has occurred, collects the relevant information concerning that event, and sends the information to the remote monitor. Each sensor is found within a type manager, and is associated with an operation (or set of operations) provided by the type manager. For example, the file system (a type manager for the file object type) may have a ReadFile event sensor located in the code performing the read operation. Other sensors, such as OpenFile, ExtendFile, PhysicalBlockRead, and ModifyProtection, may also be present in the file system. Since state changes on a file object can only occur as the result of operations performed by the file system, sensors within the file system can monitor these state changes for all file objects.

Event records always contain the name of the type manager performing the operation (if the type manager is itself an object), the event type, the name of the referenced object, and the name of the initiator. Thus, all four components of a state change are recorded for later analysis.

Event records are stored in *receptacles*, which handle the enabling and synchronization of event records. Receptacles are abstract objects in their own right, whose type manager is the resident monitor. Events are *enabled* by the resident monitor by setting switches in the receptacle. Locks in the receptacle arbitrate simultaneous access and modification of the switches by the resident monitor and the sensors.

There are several operations on receptacles supported by the resident monitor. To install a receptacle in an object, the type manager for that object requests a receptacle from the resident monitor. The type manager then places the receptacle in the object at a location determined by the type manager, which has total control over the representation of the object. To enable (or disable) an event for an object, the resident monitor presents the object to the appropriate type manager with a request for the receptacle contained in the object. The resident monitor then modifies the appropriate switch in the receptacle. Therefore, the minimum functionality a type manager must provide to support monitoring is the access receptacle operation, which can be implemented quite easily , and the install receptacle operation, which is also straightforward to implement.

Any sensor using the receptacle contained in an object must reside in the type manager for that object. When such a sensor is encountered during the processing of a requested operation, the event identification, object name, initiator name, performer name, and receptacle, as well as any additional information provided by the sensor, is passed to the resident monitor, and a store event record operation is performed by the resident monitor. First the appropriate enable switch in the receptacle is checked to ensure that the event is enabled for this receptacle. If so, an event record in the proper format is then sent to the remote monitor.

Placing the enable switches in the receptacle allows great flexibility in the enabling of events. Receptacles are associated with passive objects and processes (active entities). A receptacle associated with a passive object arbitrates the collection of monitoring information for that object. Enabling the file read event for the receptacle associated with a particular file causes event record to be collected for file reads *only for this file* by any file system process. On the other hand, enabling the file read event for the receptacle associated with a particular file system process causes event records to be collected for file reads on any file performed *only by this file system process*. An analogous selection applies to the initiator of a file read operation. The event records can thus be filtered at a fine grain along three dimensions: by object, performer, or initiator. Each sensor supports filtering of an event type in only one dimension. However, several sensors (and event types) can be associated with an event (such as file read), each designating a different receptacle to arbitrate event record generation by the sensor. Viewing the set of possible event records as a discrete four-dimensional space with axes consisting of event types, objects, performers, and initiators, the event records generated by a particular sensor form a two-dimensional plane, parallel to two of the axes, and intersecting the event axis and one other axis. The event records enabled by a particular receptacle form a series of two-dimensional planes, all parallel yet intersecting the event axis at different points[6].

To achieve higher degrees of filtering (either a line or a point in the event space) requires either additional information be to stored in the receptacle (and additional processing to determine if the event is

[6]A point in this space may include several event records, each representing the same event, object, performer, and initiator, but occuring at different times. Of course, time could be considered as yet another dimension. Visualizing the event space is more difficult with such a change; the author finds four dimensions hard enough!

indeed enabled), or new receptacles representing component pairs (such as object-performer) or component triples (such as object-performer-initiator) to be created and associated with the participating objects. Both alternatives require super-linear space and/or time, and thus are unacceptable in an environment supporting many objects.

The typing model applies to all levels of abstraction, from the hardware (with objects such as memory locations, interrupt lines, device registers), the firmware, the language level (with objects such as variables, semaphores, procedures), to the process level and the program level (with objects such as servers, databases, users). The data collection mechanism can be used at all of these levels, presenting a consistent interface to the rest of the monitor: event records containing the event type, object, performer, and initiator, as well as other, event-specific, information. By associating receptacles with the objects defined at a particular level of abstraction, the full filtering capabilities can also be realized. However, the implementation will differ greatly from level to level, and there must be ways to transmit the information gathered at the lower levels, especially at the hardware and firmware levels, to the higher levels where they can be dealt with by the monitor.

7.3. Integrating Sampling and Tracing

In the preceeding discussion, the assumption was made that the event record is sensed amd communicated to the remote monitor when the event occurs. Such event records are called *traced* event records, since their generation is *synchronous* with the event, and thus with the operation whose object, performer, and initiator is named in the event record.

Sampled event records, on the other hand, are generated at the request of the monitor, *asynchronously* with the event they are concerned with. As an example, a sensor located in the scheduler of an operating system could generate traced event records pertaining to context switching: process x started running at time t_1, process y started running at time t_2, etc. Another sensor located in the scheduler could generate sampled event records at the request of the monitor: process z is now running.

In the context of the type model, both sensors were executed as a result of an operation supported by the scheduler; the former by the dispatch operation, the latter by the report current process operation. The only distinction is the nature of the initiator--either a random process in the system, or the resident monitor. As far as the low level data collection mechanism is concerned, there is absolutely no difference between sampling and tracing: the sensor, when encountered in the course of executing the operation, checks the enable switch in the appropriate receptacle, and, if set, sends the event record to the remote monitor. Also, the resident monitor must now have the ability to invoke operations in other type managers, but the added complexity is where it should be: in the monitor rather than in the user's (i.e., type manager's) code.

7.4. Interaction with the Remote Monitor

Since the remote monitor contains a large knowledge base concerning monitoring, it is important that it be system independent. Hence, the remote monitor's view of the world is an abstraction supported by the resident monitor interacting with a particular operating system with certain assumptions being made about the event records being generated. There is a tradeoff between strong assumptions, which are difficult to support by the resident monitor, and weak assumptions, which make the derivation of high level information from event records difficult. This section is concerned with the environment as seen by the remote monitor, and how this view is supported.

The format of the event record is one aspect to be standardized. Each event record is divided into a fixed and a variable part. The fixed part is identical in format in all event records, and included the event number, the (possibly nil) names of the initiator, performer, and object, and (possibly) a timestamp. The variable part contains the domains of the event record, i.e., the additional information provided by the sensor. Domains must be formatted in a defined external type (currently either a short integer, a long integer, a boolean, a variable length character string, or a remote name (see below)), with the sensor mapping values in an internal format to an external type. Names and timestamps are much harder to standardize; the rest of this section will describe our approach to these two issues in the context of data collection.

7.4.1. Naming

There are several name spaces within the monitor for operating system objects; this section is concerned with *internal names* (the operating system specific names), *remote names* (system independent names which are processed by the remote monitor), and the mapping between these two name spaces. Other chapters will deal with the remaining name spaces active in the monitor.

Internal names allow the resident monitor to gain access to the object in question. The internal name for a file may be the disk address of its directory entry, or the inode number in the case of Unix files [56]; the internal name for a process might be a memory address of a process control block, or an offset into the process table. Object-based systems allow a consistent naming scheme to be used; the operating system supports the addressing of *all* objects by using a name in a standard format. These names are usually protected, so that processes must acquire names, rather than being allowed to arbitrarily generate them. StarOS [27] and Medusa [51], the two operating systems the monitor was implemented on, are both object-based systems; internal names are called capabilities and descriptors, respectively. A remote name is an integer which can be mapped to an internal name, thereby allowing the actual object to be referenced by the resident monitor.

It is helpful to examine briefly how names in these two name spaces are used by the monitor. When the monitor starts, it knows no names. The user issues a query, and the remote monitor instructs the resident monitor to find certain objects and to return the remote names of these objects. At that point, the remote monitor sends some of these names back to the resident monitor, instructing it to enable certain events on those objects. As these events subsequently occur, event records containing the remote names in their fixed parts are sent to the remote monitor.

In order for this interaction to occur successfully, several invarients concerning remote names must be guaranteed:

Uniqueness	Remote names must be unique (one remote name for each object (internal name)) for event records to make any sense at all.
Injectivity	Remote names must be unambiguous (one object for each remote name), for the same reason.
Bidirectionality	The mapping must be bidirectional; in particular, we must be able to find the object given a remote name.
Completeness	When an event record is sent to the remote monitor, the remote name for the object, the sensor, and the initiator must be available.
Lifetime	The mapping must allow operating system objects to be garbage collected.

Unfortunately, there is no mechanism for producing remote names which will satisfy all five invarients, although different mechanisms violate different invarients. Assume we have a remote name for an object. We can either

- not allow garbage collection, or
- allow garbage collection, and violate the bidirectionality invarient (if the object is deleted, we cannot map the remote name to an internal name), or
- violate the injectivity invarient (map the old remote name onto a new object which has the same internal name as the old, deleted object).

The second and third alternatives apply to operating systems that do not support a consistent, protected internal name space. For example, the disk address of a file may be a perfectly valid name for the file while it exists (assuming the directory is not reorganized). However, there is no guarantee that the file will not be deleted and the entry replaced with that of another, newly created file.

The approach used here applies to object-based operating systems, and will support capability addressing at the expense of a slightly corrupted bidirectional invarient: sometimes the mapping from the remote name to the internal name will not work. To see how this is done, we must first explain the mapping between internal and remote names.

Remote names will be of the form <epoch><minor name>. The <minor name> will be formed from the internal name in a system-dependent fashion (this does not circumvent protection, since the internal name may be *read*, but not *written*). Whenever an event record is constructed, the current <epoch> for each <minor name> in the event record will be concatenated with the <minor name>, thereby forming a remote name.

The resident monitor will maintain a list of internal names it has collected from the event records it has sent to the remote monitor. Whenever a remote name is sent to the resident monitor, the associated internal name will be retrieved using the <minor name> field. Whenever an object is deleted, the <epoch> for the <minor name> for that object be incremented, to ensure that the next object created with this internal name will be mapped to a unique remote name. Note that interaction between the garbage collector and the resident monitor is required to keep the <epoch> values consistent.

The mechanism outlined above satisfies all of the invarients except the garbage collection invarient. As long as an internal name resides in the resident monitor, the object it refers to cannot be garbage collected. Having many internal names in the resident monitor imposes an unnecessary processing and memory burden on the system. Therefore the mechanism must allow internal names to be removed from the resident monitor.

Stated simply, an internal name should be removed if it will never be needed again[7]. Since it is impossible to predict when this will be the case, various approximations may be used:

- the object has an explicit destroy operation performed on it;
- there is only one internal name (the resident monitor's) referencing this object (and thus, the object is nonexistent as far as the rest of the environment is concerned);
- the remote monitor will never send the remote name to the resident monitor in a request;
- it has been a long time since an event record has been generated;

[7]Put another way, all the invarients can be satisfied given an omniscient resident monitor!

- the user has specified that this object is unimportant (or equivalently, has not specified that this object is important);
- the object is of an unimportant type (or equivalently, is not of an important type);
- the remote monitor doesn't have a remote name for this object.

The first alternative applies only to objects which have an explicit destroy operation on them which can be monitored. The semantics of the destroy operation from the monitor's point of view is that, after the operation has been performed, nothing interesting will ever happen to this object, i.e., no event records containing a name for this object will ever be generated. The second alternative depends on a garbage collector which can determine whether there is only one internal name extant; current garbage collectors do not have this capability < is this true?> The last four heuristics depend on psychological aspects, and can be shown to be appropriate only after experimentation with many users over long periods of time.

7.4.2. Time

Time is a difficult problem when monitoring a distributed system. There must be a mapping from the local time recorded in the event records to global time. Unfortunately, it is theoretically impossible to exactly sychronize imprecise physical clocks over a geographical network with indeterministic transmission times. A more practical constraint is keeping the overhead incurred in synchronizing the local clocks acceptably low.

7.5. Sensor Specification

During the initial development of the low level event collection mechanism, it became apparent that there were several procedural difficulties in the placement of sensors in the various operating systems. One difficulty was the that the sensor routine (which stores the event records) was becoming quite cumbersome. One design required twelve parameters for a typical sensor involving three domains! Since the sensors were to be placed in critical portions of the operating system, there was little room for error in the specification of these parameters. A second problem was the assignment of event numbers; an incorrect event number in a sensor routine would result in the absence of event records of that type--a situation that might be difficult to detect by the user interacting with the remote monitor. Two sensors with the same event number would cause havoc within the remote monitor. A third problem is maintaining consistency between the remote monitor's view of the world and the world as it actually is. This is especially true during the early development of the monitor, when the collection of sensors inside the operating system, and the various attributes of those sensors, is changing frequently.

Finally, all these problems are exacerbated by the sheer number of sensors: we can easily envision several *hundred* sensors in an operating system when it has been fully instrumented. The task of ensuring that ali of these sensors, which are distributed across many source files, are correct and consistent, both between each other and with the data structures within the remote mo^nitor, is unmanageable if it remains a manual one.

7.5.1. User-Defined Events

All of the issues mentioned above are also present when users are allowed to define events. Several, including the event number assignment and communicating the sensors' specifications to the remote monitor, become even more difficult in the presence of user-defined events.

In general, the less the user has to specify, the less that can go wrong with the specification. In a best case senario, the user would specify the interesting events and indicate where the sensors for these events were to be placed in the code. The sensor would be produced automatically from the specifications, and would be as efficient as one crafted by hand. When the program was run, the event types generated by these sensors would automatically be defined as relations with the full query language available for manipulating events generated by the sensor. In addition, enabling and disabling of the sensors in the user's program would be handled automatically as a side effect of evaluating queries referencing these relations.

The solution, described in this section, is to create a database, called the *sensor description file (SDF)* containing information on the sensors defined in a given taskforce (a *taskforce* is a collection of processes cooperating to perform a particular task [28]). The sensor description allows the above senario to be realized in its entirety: placing sensors in a program involves merely creating a sensor description file consisting of a few lines per event type and object type, adding the sensors to the program (one line per sensor), and running a program (called the *description file preprocessor* (*DFPre*)) with the SDF as input. All of the details are automatically taken care of by DFPre and by the monitor.

7.5.2. Syntax and Definitions

A sensor description file co..sists of a set of *objects* partitioned into *classes*. Each object is associated with a set of class-dependent *attributes*. There can be one or more *values* for each attribute, and some attributes can have objects as values. The syntax follows this description quite closely:

```
<description file>        ::= <objects>

<objects>                 ::= <object> | <object> <objects>
<object>                  ::= "(" <class> <attribute list> ")"
```

```
<attribute list>          ::= <attribute> | <attribute> <attribute list>
<attribute>               ::= "(" <attribute name> <attribute values> ")"

<attribute values>        ::= <attribute value> | <attribute value> <attribute values>
<attribute value>         ::= <object> | <object name> | <atom> | <integer> | <string> | <list>

<class>                   ::= <atom>
<objectname>              ::= <atom>
<attributename>           ::= <atom>
```

As an example, the following description file has one object with two attributes:

```
(event     (name Iteration)
           (timestamp true)
)
```

An SDF describes the sensors defined in a given taskforce. Since the operating system is itself a taskforce (or collection of taskforces), one SDF specifies the sensors embedded in the operating system. The **Taskforce** class includes attributes which hold for the task force as a whole. The **ObjectType** class describes the objects which can be monitored by sensors defined in the SDF. The **SensorProcess** class contains those attributes relevant to a particular process (i.e., a type manager). The **Event** class contains most of the attributes, including the following:

Location	the sensor process containing the sensor for this event;
Object	the object type this event refers to;
Timestamp	whether timestamps are to be included in the event record;
MinorType	how the event is to be triggered;
SpaceTimeRatio	the relative tradeoff between space and time efficiency in the sensor;
Domains	the domains included in the event.

The **Domain** class includes attributes relating to each domain, particularly the type attribute.

7.5.3. The Description File Preprocessor

The description file preprocessor (DFPre) reads in a description, performs syntactic and semantic checking, and outputs one or more files containing information derived from the input file. The position of DFPre in the program development process is illustrated in figure 34.

The require files contain routine definitions to be used by the sensors. For example, each event results

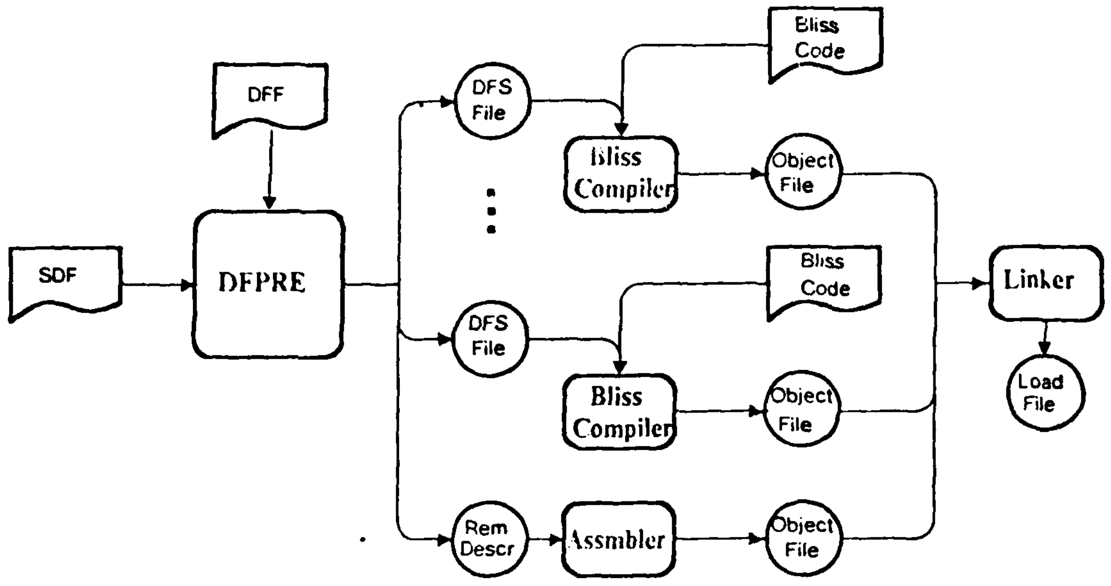


Figure 34: The position of DFPre in the program development process.

in the definition of a sensor routine. If the SDF contained the following event class (a fragment of an actual SDF used to monitor a parallel partial differential equation (PDE) program):

```
(event      (name Iteration)
            (location PDESolver)
            (timestamp true)
            (domains (domain (name IterationNumber)
                             (type Integer)
            )
            ...
)
```

then the routine IterationSensor, with one parameter (IterationNumber) would be defined. To place a sensor for this event, the user would simply put the line

```
    IterationSensor(ThisIterationNumber)
```

in the PDESolver code, where ThisIterationNumber is a variable containing the current iteration number. The require files contain virtually all the details necessary for the monitor to interact with this sensor.

The remote description is a specially-formated file containing the information in the SDF of use to the remote monitor. This file is assembled and loaded with the user's program. When the resident monitor encounters the taskforce, it ships the remote description to the remote monitor as a series of event records. When this operation completes, the remote monitor is aware of the events, sensor processes, and object types defined in the task force, and knows how to enable and disable the events.

This mechanism also works for the SDF(s) associated with the operating system. When the remote monitor is started, it knows of no events, sensor processes, or object types. The resident monitor, when started, first establishes contact with the remote monitor, then sends over the remote description, thereby defining the events contained in the operating system.

In addition to rectifying the problems introduced on page 114, using SDF's has several other advantages. Since DFPre has detailed information on the structure of each event, the code for that sensor can be tailored precisely to that event. The spacetimeratio attribute is especially useful in this regard. DFPre can also collection aggregate information concerning, for instance, all events located in a particular process, in order to perform global optimization. The receptacles for each object type can also be configured quite precisely. And finally, the information in the remote description can be used to compensate for resources consumed during event collection, since the remote monitor will know what processing was involved in storing each event record.

7.6. An Example

To illustrate the actions of the monitor, we will examine how a particular program running on Cm* is monitored. The program solves Laplace's partial differential equation with given boundary conditions (Dirichlet's problem) by the method of finite differences. The equation

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial x^2 u(x,y)}{\partial y^2} = 0$$

is solved for points on an m by n rectangular grid, where only the values at the outer edges of the grid are given. The solution is found iteratively. On each iteration, the new value of each element is set to the arithmetic average of the values of its four adjacent neighbors.

Several processes and several processors work on the grid simultaneously. The grid is partitioned into regions, with one process responsible for each region. The configuration is shown in figure 34. Note that the solvers require access to adjacent regions to derive new values for points on the boundary of their region.

There are many possible ways to synchronize the processes. The most efficient is the purely asynchronous method. The processes are only synchronized at the beginning of the computation. This means that, due to differences in the scheduling and in the data that each process is working on, some processes may perform many more iterations than others.

The proposed experiment will investigate the relative synchrony of two of the processes operating on adjacent regions. If one of the processes (call it P_1) gets behind the other process (P_2), then the second

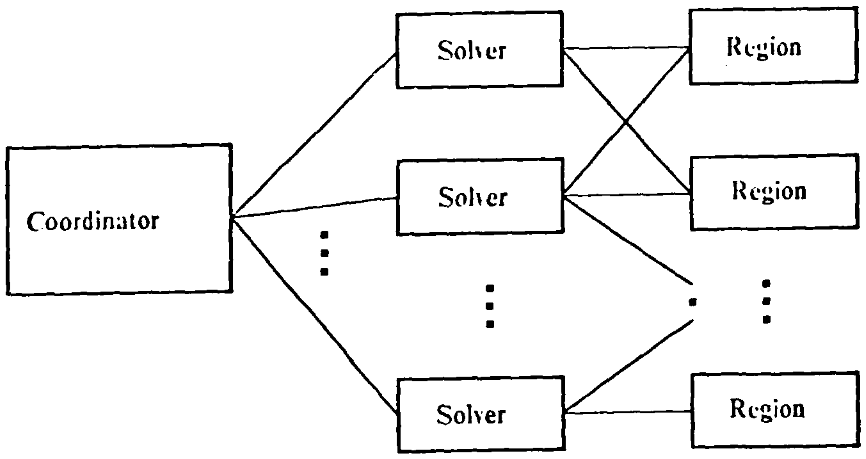


Figure 35: Configuration of the PDE task force

process will be using older values for the points on the boundary, possibly slowing the convergence for the entire grid. This experiment will focus on those periods of time when P_1 gets significantly behind P_2 (i.e., more than one iteration).

One sensor is needed, a traced event sensor which generates an event record each time the solver process begins a new iteration. The sensor descriptor file is shown in appendix PDESDF.

When the PDE program is loaded onto Cm*, the monitor is sent the information provided by the SDF. Since the Iteration sensor has a minortype of sensortraced, the events generated by this sensor are automatically converted into a primitive period relation by the monitor. Thus, when the remote monitor receives the SDF describing the Iteration sensor, the primitive period Iteration is defined, with two domains: Process, the name of the process generating the event record, and IterationNum, an integer designating the iteration which has just begun. Derived relations can now be specified using the Iteration relation, as will be described in detail in the next chapter.

7.7. Summary

This chapter first presented a model for the environment the data collection mechanism was to execute in, the type model, and then a mechanism which integrates well with this model. The occurrence of an event is tied to four components: the operation, the object being operated on, the performer of the operation, and the initiator of the operation. These components are recorded in the event record generated by the sensor, along with additional information germane to the event. A new type of object, the receptacle, was introduced to arbitrate the generation of event records, and great flexibility in filtering was shown to be possible by associating the receptacles with the various entities participating in the event. We demonstrated that, from the point of view of the sensors, there was absolutely no difference between traced and sampled event records, the distinction lying instead in the identity of the initiator of the operation involving the sensor. Several issues involved with the interaction of the remote monitor, in particular naming and time, were discussed, and techniques were developed for coping with the problems inherent in those areas. Finally, the substantial software engineering issues involved in placing sensors were dealt with quite successfully by having the user create a database, the SDF, which allowed the monitor to aid in installing sensors by handling all of the details for the user.

At this point, it is possible to answer the query of chapter 2,

Problem 5: Is it possible to provide effective data collection mechanisms?

with a resounding Yes!:

Result 5: The system-dependent aspects of data collection can be embedded entirely in the primitive relations. The user does not have to specify how the data is collected, or to worry about the details of collecting, formatting, and processing the event records. Sensors are efficient, and easy to install.

8. Simon: A Simple Monitor for StarOS

Richard Snodgrass

8.1. Introduction

This document specifies the user interface and overall design for an initial implementation of a monitoring system for StarOS. The reader should read my thesis proposal, which gives an overview of the relational approach to monitoring, before reading this paper.

8.1.1. Objectives

The objectives of this first implementation are three-fold: (a) to test my ideas on a significant set of monitoring tasks; (b) to delimit the various issues involved in monitoring; and (c) to get experience with the VAX and StarOS programming environments before attempting the design of a comprehensive system.

There were several limitations on the scope of this implementation in order to complete it in a reasonable amount of time. Although the thesis is based on the monitor supporting the conceptual model of a relational database, this implementation will concentrate on the more concrete issue of dynamic incremental updating of temporal relations. The following issues will *not* be included in this implementation:

- static relations; all computations will be done in "real-time"
- quel-like query language; a relational algebra will be used instead
- efficiency; no tuning of the relational operators will be done
- incomplete information, although a few aspects may be included if they are straight-forward

Chapter 8.5 examines how the implementation discussed below might be extended to incorporate the above issues.

8.1.2. A Paradigm for the Implementor

Incremental update algorithms for temporal relations accept information in the form of "this relationship between these entities became true at time t_1" and "the relationship subsequently became false at time t_2", and use this information plus stored information concerning the current state of the relation to derive an updated relation. Relations thus evolve in time through tuples (rows of a relation) being added and removed. These changes cause relations derived from a relation to acquire or lose tuples of their own, a process continuing until the new information has been completely assimilated by the relations defined in the system. It is thus natural to concentrate on the flow of tuples (both being added and being removed) among the relations that are associated with each other through derivation expressions. Examining the monitoring concepts of primitive relations, derived relations and relational operators from this viewpoint results in rather different conceptualizations of these fundamental notions.

It is important to remember that two radically different paradigms are at work in this design. One paradigm was introduced in my thesis proposal: the process of monitoring is profitably conceptualized *by the user* as the accumulation of monitoring information in the form of relations which can then be manipulated conveniently by the user. The paradigm introduced above is rather different: the process of monitoring is profitably conceptualized *by the designer (implementor) of the monitoring system* as the creation and execution of a network of nodes across which flows information collected by the monitor. The former paradigm was discussed to some length in my thesis proposal; this document will investigate the implications of the latter paradigm.

8.1.3. Overall Design

The monitor will provide a collection of *access nodes* and *generic operator nodes*, which the user will instantiate and link together in a tangled tree. Information in the form of tuples (a collection of domains, or, alternatively, a row of the relation) will flow out of the access nodes (which will communicate with the access module(s) on Cm*) and up through the tree. Operator nodes will take tuples from one or two lower nodes and produce tuples which will be sent further up the tree. The entire hierarchy will be driven by tuples originating in the access nodes (these tuples can represent either events which occurred or samples which were automatically collected by the access module).

One might visualize this network by thinking of the nodes, both access and operator, as integrated circuits (ICs), having zero or more input pins, one or more enable pins, one or more output pins, and a data-out pin. Each pin represents a domain; a collection of input or output pins together with the enable or data-out pins signify a tuple arriving at the device or being generated by the device. For those operator ICs (nodes) requiring two tuple inputs, there would be two sets of input pins (with two enable pins). A network is constructed by selecting the appropriate ICs and ensuring that every pin of each IC is connected to at least one pin of another IC in a hierarchical fashion. Access ICs are somewhat unusual in that they are connected to the outside world. When something changes in the outside world (the "something" can be specified by the input pins) the access IC places a value on each of the output pins and raises the data-out pin, thus generating an output tuple. This triggers the IC(s) connected to this IC, causing them to place a tuple on their output pins and raise their data-out pins.

In the final design, the monitor will automatically construct the network given the high level queries specified by the user. In the meantime, the user will be provided with commands to manipulate the network (see chapter 8.4).

This network approach emphasizes the flow of information from the access module on Cm* to the user's terminal. Relations are not explicitly represented in the system; instead a relation consists of all the tuples which appear at the output pins of a particular IC (node) over the time period the node is in the network (see section 8.5.2.1 for a way to support static relations using this framework). The instantaneous snapshot of each relation is contained in the internal state of the node computing that relation. The algorithms contained in the operator nodes, while performing standard relational operations such as join and projection, are quite different from their database counterparts, since they are tuned for incremental update of temporal relations. Since relations are represented only implicitly, a mechanism for viewing a relation is vital. Among the collection of operator nodes is a display node, which merely displays the tuples it has received on its input pins.

8.1.4. A Series of Implementations

The design is evolutionary, in that it consists of a series of implementations, each extending the previous implementation and attacking a larger set of issues. The four implementations are listed below (the terms will be discussed in later chapters):

- Step 1: a truly minimal system
    - representation of tuples
    - a few generic operator nodes
    - debugging facilities, including pseudo-access nodes (which do not access StarOS but instead are used for debugging)
    - primitive constructors (create, link, unlink)
    - control mechanisms for moving tuples around the network
- Step 2: a minimal StarOS monitor
    - additional operator nodes
    - communication with StarOS
    - a few actual access nodes
- Step 3: add naming and improve the user interface

- o defstruct
- o full communication with StarOS
- o a full complement of access nodes

- Step 4: round out the system (may never be done)
    - o uncertainty considerations
    - o display of multiple relations

I expect each step to take approximately two weeks.

The monitor will be written in Franz Lisp, running on the VAX, and the access modules will reside in a single process written in Bliss/11 and run under StarOS. The two processes will communicate over the Ethernet using the Pup protocol.

8.2. Access Nodes

The access nodes collect all the information accessible by the monitor. Each node conceptually takes zero or more input domains (pins) specifying an entity and augments these with additional domains describing that entity. Thus there is an output pin for every input pin, plus the output pins computed by the access node. Each domain is typed (unlike real ICs); see Appendix I for a discussion of the allowable types and their representations. Each access node is associated with an accessing mechanism which specifies how the information in the additional domains is to be computed. Access nodes can be instantiated by the create operation (see section 8.4.1) enabling several copies to exist simultaneously in the network.

The particular access nodes that are supported were chosen for several reasons. Naming is a basic facility which should be provided early in the implementation, since the other access nodes depend on names being associated with the entities involved in the relationships accessed by the nodes. Over half of the access nodes support naming either directly or indirectly. Since processes are so prominent when first monitoring a system, about one fourth of the access nodes involve processes. To support the proposed synthetic workload generator, the TimeStamp access node was provided. And finally, several access nodes provide general information concerning modules, processors, and capabilities. The access nodes listed below allow a full range of issues concerning the collection of monitoring information to be investigated.

The description of each access node consists of a signature (the node name and the domains), a description of each domain, and the associated access mechanism for the node. Implicit in each description is the time domain (an output pin) and the enable and data-out pins. The underlined domains in the signature are both input and output pins (connected together inside the node); the domains which are not underlined are computed by the node and are thus only output pins.

8.2.1. CapaInfo (<u>ThisCapa</u>, Type, Cluster, Cm)

Domains:

ThisCapa	CAPA, specifies the capability pointing to the object described by this relation
Type	ENUMERATED: (BasicObject, KernelPO, UserPO, DeviceObject, ShadowObject, DataMailbox, CapabilityMailbox, DirectoryObject, DequeObject, StackObject)
Cluster	INTEGER, the cluster this object resides in
Cm	INTEGER, the processor this object resides in

Access Mechanism:

*** object manager request *** The type domain is computed using Basic.DFS.

8.2.2. GetMpx (<u>Nucleus</u>, Mpx)

Domains:

Nucleus	CAPA to the nucleus module
Mpx	CAPA to Mpx Basic status object

Access Mechanism:

Get the Mpx Basic status object via ***.

8.2.3. Library (<u>Index</u>, Module)

Domains:

Index	ENUMERATED index into StarOS Library
Module	CAPA of module

Access Mechanism:

Convert the index into an integer (using ModLib.DFS) and access the StarOSLibrary capa.

8.2.4. ModuleAttributes (<u>Module</u>, ModuleId, NumberOfFunctions, NumberOfProcesses)

Domains:

Module	CAPA to a module object
ModuleId	INTEGER
NumberOfFunctions	INTEGER
NumberOfProcesses	INTEGER, the total number for this module

Access Mechanism:

Sample when needed, using ABlock.DFS. Access the NumberOfProcesses using ProcessSetTotalCount in ProSet.DFS. Additional output domains will be added if necessary.

8.2.5. ModuleFunctions (<u>Module</u>, Function, NumberOfProcesses, InitStackSize)

Domains:

Module	CAPA
Function:	INTEGER
NumberOfProcesses	INTEGER, for this function
InitStackSize	INTEGER

Access Mechanism:

Sampled when needed, using function attribute block of module, one for each function, using ABlock.DFS. Access NumberOfProcesses using ProcessSetFunctionCount in ProSet.DFS. Additional output domains will be added if needed.

8.2.6. ModulesLoaded (<u>UserInterface</u>, Module, Name)

Domains:

UserInterface	CAPA to user interface process
Module	CAPA
Name	STRING name of Module

Access Mechanism:

Sampled when needed, using data structures found in the UIModuleRock capa (User[x335ID20]); look up string in $ModuleTable (UserCm.B11[x335ID20]). ***

8.2.7. ModuleProcs (<u>Module</u>, Function, Process)

Domains:

Module	CAPA
Function	INTEGER
Process	CAPA

Access Mechanism:

Sampled when needed; grab first process of this function using ProcessSet (Modul.DFS, ProSet.DFS); for each process (next = EnvBrotherProcess in Enviro.DFS) assign to process.

8.2.8. MpxStatus (<u>MpxObject</u>, CurrentProc)

Domains:

MpxObject	CAPA
CurrentProc	CAPA

Access Mechanism:

Sample with high frequency, using ***.

8.2.9. ProcessAttributes (<u>Process</u>, IsActivated, RBRD, CPII, Cluster, Cm, FunctionNumber, Module)

Domains:

Process	CAPA
IsActivated	BOOLEAN
RBRD	ENUMERATED: (Ready, Blocked, Running, Done)
CPII	ENUMERATED: (Created, PartInitialized, Initialized)
Cluster	INTEGER
Cm	INTEGER
FunctionNumber	INTEGER
Module	CAPA

Access Mechanism:

Sample when needed, by accessing data and capa part of process object, using StatVe.DFS and Enviro.DFS. Access Cluster and Cm using ***.

8.2.10. ProcCapas (<u>Process</u>, Occupant)

Doains:

Process	CAPA
Slot	INTEGER
Occupant	CAPA

Access Mechanism:

Sampled, using UENVNameSpace (in Enviro.DFS).

8.2.11. ProcessorMpx (<u>Mpx</u>, Cluster, Cm)

Domains:

Mpx	CAPA to Mpx Basic object
Cluster	INTEGER
Cm	INTEGER

Access Mechanism:

Sampled when needed, using ***.

8.2.12. ProcessorUniverse (<u>Configurator</u>, Cluster, Cm, MemorySize, Status)

Domains:

Configurator	CAPA to module
Cluster	INTEGER
Cm	INTEGER
MemorySize	INTEGER ***
Status	ENUMERATED: (CmMissing, CmSuspect, PcDown, CmAvailable, BootCm)

Access Mechanism:

Sampled when needed, using Config.DFS.

8.2.13. TimeStamp (Type, SubType, Args)

Domains:

Type	***
SubType	***
Args	***

Access Mechanism:

Traced by receiving a message from the application.

8.2.14. UserId (<u>UserInterface</u>, UserName)

Domains:

UserInterface	CAPA to user interface process
UserName	STRING

Access Mechanism:

Sample when needed, using ***.

8.3. Generic Operator Nodes

In order to derive new relationships that are in a more useful form than the relationships expressed by the access nodes, it is necessary to use operations that can be performed on the tuples from one or two relations to compute a new relation. The system provides a collection of *generic operator nodes* which can be instantiated by specifying the parameters associated with the node. In the descriptions that follow, the parameters syntactically follow the name of the node. An integer is used to specify a domain (domains are ordered, with the first domain being domain 1; the implicit time domain is always domain 0). Parameters make operator nodes more convenient than ICs, since they can specify which input pins will be used to compute the function and often which function is computed as well. The description of each generic operator node consists of a signature (the node name, the instantiation parameters in brackets and the output domains in parentheses) and an explanation of the computation performed by the node. Some nodes append new output domains computed from existing domains; the rest map the input domains to the output domains in some fashion. The names of the output domains not computed by the node are taken from the input domains they are connected to. Note that all of these operations rely heavily on the implicit time domain in their computations.

8.3.1. ApplyOp [Operation Domain$_1$ Domain$_2$] (Result); unary

Computes the domain Result whose value is the result of the specified operation on the specified domains. The operation can be any Lisp function which can be applied to two arguments. Special semantics are associated with arithmetic operations on temporal domains; see Appendix IV.

8.3.2. Constant [Constant] (Constant); no input tuples

Generates a constant on its Constant output pin. This constant can be of any type.

8.3.3. Count [MajorDomain MinorDomain] (Count); unary

Computes the count of the tuples having the same value for the MajorDomain after a projection on the MajorDomain and MinorDomain has been done (i.e., two tuples with the same MajorDomain and MinorDomain count only as one).

8.3.4. Display [Name_1 ... Name_n]; unary

Maintains a display of the argument relation on the screen. The instantiation parameters are column (domain) titles.

8.3.5. Duration (Duration); unary

Computes the TEMPORAL domain Duration whose value is the length of time (in milliseconds) the tuple was valid.

8.3.6. Join [Predicate Domain_1 Domain_2]; binary

The output tuple contains all the domains of the two input tupl[illegible] which can contain a different number of tuples. Each time a tuple enters the left input, it is concatenateu with all of the tuples that have entered into the right input. The predicate is then applied to the corresponding domain_1 coming from the left tuple and domain_2 coming from the right tuple. If the predicate returns non-nil, the concatenated tuple is placed on the output pins. An analogous process occurs each time a tuple enters the right input.

8.3.7. Projection [NumDomains Domain_1 ... $\text{Domain}_{\text{NumDomains}}$]; unary

The projection node is conceptually a parameterized cross-point domain switch. The values of a particular subset of the domains in the input tuple are selected and places in the specified order on the output pins. An example is

```
    Projection [6 4 3]
```

which takes the third and fourth input pins and connects them to the second and first output pins, respectively. Input pins 1, 2, 5, and 6 get connected to output pins 3, 4, 5, and 6 respectively. Note that the first NumDomain input pins are connected to a like number of output pins and that the information from the rest of the input pins is discarded.

8.3.8. Selection [Predicate Domain$_1$ Domain$_2$]; unary

If the predicate returns non-nil, then the input pins are copied to the output pins.

8.3.9. SelectConstant [Predicate Domain Constant]; unary

Similar to the Selection operator, except that the third argument is a constant rather than a domain index.

8.3.10. Sum [MajorDomain MinorDomain] (Sum); unary

Similar to Count, except the sum of the values in MinorDomain (which must be an INTEGER or TEMPORAL domain) is computed, rather than the number of distinct values.

8.4. Node Interconnection

In previous chapters the access nodes and generic operator nodes supported by the system were described. The access nodes provide all the information the monitor knows about the application running on Cm*; the operator nodes are used to convert this information into a form more useful to the user. This is done by combining the nodes into a *tangled tree* with the links corresponding to paths over which individual tuples flow.

The analogy between nodes and integrated circuits breaks down somewhat when one examines how the nodes are interconnected. Ideally (?) one would like to specify all connections between pins in the network. The difficulty lies in the fact that the pins represent domains of the relation, yet the implementation deals only with *tuples*, which are collections of domains. This is natural, since we are still dealing with relations. A second diffic ty with the analogy is that each instantiated node has a unspecified number of input pins which are connected *directly* to an *equal number* of output pins (unlike standard ICs!). The exact number of pins is determined when the node is connected into the network. The reason is that a node should take as arguments relations containing an arbitrary number of domains, rather than having, say, a generic operator node called Duration1 which computes the duration of all one domain relations, another node called Duration2, etc. One way to deal with this problem is to include the *number of domains* as an instantiation parameter. Instead, the manner in which the pins are connected is constrained. The set of output pins of each device is grouped together into either a single tuple or a left and right tuple (the nodes can have 0, 1, or 2 input tuples). The link command is provided to connect the output tuple of one device to an input tuple of another device. The projection operator node, which maps a number of input domains (the input tuple) into a (possibly smaller) number of output domains (the output tuple), provides some of the flexibility that the link command removes.

8.4.1. Primitive Constructors

There are three primitive constructors. The first creates an instantiation of an access or operator node. The second creates a link between two nodes, and the third destroys an existing link. Given the predefined nodes and these constructors, it is possible to specify arbitrarily complex computations on monitoring data.

8.4.1.1. (Create name newname (arg_1 ... arg_n))

This operation creates an instantiation of the node *name* with the specified instantiation parameters and associates the name *newname* with it. For example,

```
    (Create 'ApplyOp 'Add12 '(+ 1 2))
```

creates the node **Add12**, which will add the values of the first two domains and append the result as an additional domain of the tuple. Instantiation parameters are only allowed for operator nodes. The instantiated node is assigned to *newname* and the create operation returns *nil*. Note that the number of pins has not yet been determined (although some restrictions may have been placed on them; for instance, **Add12** has at least two input and three output pins).

8.4.1.2. (Link fromnode 'onode)

This operation connects the output tuple of the node *fromnode* to an input tuple of *tonode*, which must have at least one input tuple. For binary operator nodes, use

```
    (Link fromnode tonode left)
```

or

```
    (Link fromnode tonode right)
```

The link operation must be able to determine how many output pins there are on *fromnode* (how this is done will be seen in a moment). It then connects all the output pins of *fromnode* (the output tuple) to the selected input pins (the input tuple) of *tonode*. At this time, the number of output pins on *tonode* can be determined by adding the number of domains computed by *tonode* (which appear *after* the output domains connected directly to the input domains) to the number of input pins coming from the output tuple of *fromnode*. Note that this requirement forces the network to be a fully connected dag, or tangled tree. This property is satisfied by all networks which can be derived from relational queries.

Once the number of input and output pins has been determined, the link operation merely links them together in the order in which they occur (output pin 1 of *fromnode* is connected to input pin 1 of *tonode*, etc.)

8.4.1.3. (Unlink fromnode tonode)

If these nodes have been connected using the link command, then this link is removed. Otherwise, this operation has no effect.

8.4.2. Two Examples

To illustrate the link operation, suppose that

```
(create C Constant (1))
(create A ApplyOp (+ 1 1))
(create L Library)
```

had been executed. The instantiated node C has at least one output domain (called Constant); A has at least one input domain and two outpu′ ɔomains (one called Result); and L has at least one input domain and at least two output domains (one called Module). All nodes have one input tuple and one output tuple. After executing

```
(link C A)
```

C has no input domains and one output domain (called Constant); A has one input domain called Constant and two output domains, Constant and Result. Proceeding further,

```
(create P Projection (2 2 1))
(link A P)
(link P L)
```

P simply interchanges the Constant and Result domains, so that the Result domain can be used as an Index to L. L has two input domains, Result and Constant, and three output domains, Result, Constant, and Module. Note that the input domain specified in the signature for Library was assumed by the Link command to be the first input domain.

As a more extended example, suppose we wanted a list of all the processes owned by any given user. Informally, this list can be derived by getting the user interface module out of the StarOS library, going down the user interface processes accessible from this module until we find the user's user interface process, getting all the modules loaded by the user, and then getting all the processes which are invoked functions of those modules (whew!!). This process is illustrated below:

```
Constant[UserInterface]
          |
          |
Library (Index, Module)
                |
                |
ModuleProcs (Module, Function, Process)
                                  |
                                  |
           UserId (UserInterface, UserName)
              |
              |
           Select[UserName = RS41] (UserInterface, UserName)
                                           |
                                           |
             ModulesLoaded (UserInterface, Module, Name)
                                              |
                                              |
                        ModuleProcs (Module, Function, Process)
                                                |         |
                                                |         |
```

First a few comments on the notation will be given. Instantiation parameters are enclosed in square brackets; output domains are enclosed in parentheses. The vertical lines between domains indicate the connection of those domains; the rest are eliminated by projection nodes, which are not shown. Vertical lines between nodes indicate *all* of the output domains of the upper node are connected to input domains of the lower node. To express this network in terms of the operations given above, the projection nodes and the connections must be made explicit:

```
(Create UI Constant (UserInterface))         ;UserInterface is ENUMERATED
(Create Lib1 Library (UserInterface))        ;get the user interface module
(Link UI Lib1)
(Create Proj1 Projection (1 2))              ;the module domain
(Link Lib1 Proj1)
(Create MP1 ModuleProcs)                     ;get the user interface procs
(Link Proj1 MP1)
(Create Proj2 Projection (1 3))              ;the process domain
(Link MP1 Proj2)
(Create User1 UserID)                        ;get the user ids
(Link Proj2 User1)
(Create SC1 SelectConstant (EQ 2 RS41))      ;UserName = RS41
(Link User1 SC1)                             ;find correct interface proc
(Create Proj3 Projection (1 1))              ;user interface process domain
(Link SC1 Proj3)
(Create ML1 ModulesLoaded)                   ;get the modules loaded by
(Link Proj3 ML1)                             ;this user
(Create Proj4 Projection (1 2))              ;the modules domain
(Link ML1 Proj4)
(Create MP2 ModuleProcs)                     ;get the processes of these
(Link Proj4 MP2)                             ;modules
(Create Proj5 Projection (2 1 3))            ;the process and
(Link MP2 Proj5)                             ;module domains
(Create D1 Display (Modules Processes))      ;display them
(Link Proj5 D1)
```

There are several problems with this notation. First, it is quite verbose. Although there are only seven "useful" nodes (as illustrated before), constructing the network requires twenty-five speparate operations. Second, it necessitates inventing names for the instantiated nodes (one way to get around this would be to have the create operation automatically invent a name for the newly instantiated node). Third, it ignores the names of the input and output pins, instead relying on the implicit ordering enforced by the link operation and the use of integer constants for domain indices. Finally, there is no way to parameterize the entire subnetwork, or to remove it conveniently. In short, the notation is at too low a level. Although it provides the abstraction from pins to collections of pins (tuples), it constructs an network rather than specifying operators that take relations as arguments. The next section discusses a notation which solves most of these problems.

8.4.3. Defining Structures

The *defstruct* operation translates a higher-level description of a network structure to an internal form which an extended create operation then transforms into a list of create and link operations. It makes certain assumptions concerning the form of the network to be constructed, and thus is less general than the create and link operations themselves. The defstruct operation is intended to fit between a high level calculus-based query language such as Quel and the low level create and link operations just discussed. The transformation from a query to a defstruct specification should be relatively straightforward; the same can be said for the transformation performed by defstruct. Thus the defstruct syntax is designed to appear as much as possible to be relational operators (in the form of generic operator nodes) operating on relations (in the form of output tuples of nodes at the top of trees within the network) to produce relations.

The overall syntax of the operation is

```
(defstruct newname (param₁ ... paramₙ) (domain₁ ... domainₙ)
        source)
```

The defstruct operation creates a *structure* which is similar to a generic operator node with no inputs in that it is instantiated with a create operation with zero or more instantiation parameters. The list of domains specifies the names of the output pins for the structure. The source is an expression which states where the information that is routed to the output pins comes from. It has the following syntax:

```
(node (param₁ ... paramₙ) (domain₁ ... domainₖ)
        leftsource rightsource)
```

The node specifies an access or generic operator node (not one that has been created). The list of domains specifies which domains will be provided by this source. Conceptually, the domains provided by leftsource (and rightsource if the node is binary) are linked to the node (after it has been created) as input tuples in the order they are specified in leftsource and rightsource. The names of these domains are also given to the names of the output domains to which they are connected. These output pins are connected to a projection node which selects the domains named in the domain list. Thus each source specification is translated into an operator or access node-projection node pair, with generated names for the instantiated components. To illustrate, the following structure

```
(defstruct UserProcess (UName) (UName ModuleName Process)
        (ModuleProcs (Module Process)
                (ModulesLoaded (Mods)
                        ((Select (EQ Id Name)) (Module)
                                (UserID (Module Id)
                                        (ModuleProcs (Procs)
                                                (Library (Module)
                                                        Constant (UName)
                                                ]
```

is identical to the one specified earlier when

```
(create UserProcess (RS41))
```

is executed.

Note that the syntax for source bears a strong resemblance to the defstruct syntax. The primary difference is that the arg_i in defstruct are the *names* (i.e., the formals) of instantiation parameters which are used within the structure being defined, whereas the arg_i in source are the *values* (i.e., the actuals) of the instantiation parameters which are provided to the node. This correspondence allows the source to be the name of a structure (which appears in an earlier defstruct) followed by the values of the instantiation parameters and the names of the domains (nothing should follow the list of domains when a structure instantiation is used as a source). Using a structure as a source in another structure implies either that defstruct be able to treat this structure as a special case or that the link operation be able to handle structures. The latter change will be made, with the added provision of allowing unlink to undo the changes caused by the corresponding link operation.

8.5. Conclusion

This report has outlined an evolutionary (as opposed to revolutionary?!) approach to implementing a system supporting incremental updating of temporal relations.

8.5.1. Abstraction

The facilities provided by the monitor has been presented starting at a very low level of abstraction and moving up to a middle level of abstraction: The initial conceptual view was one of ICs (nodes) connected in a network. Each IC had a number of input, enable, and output pins, and one enable pin. This view allowed the primitive nodes provided by the monitor to be defined. After the various generic operator and access nodes were listed, a set of three primitive constructors were described which perceived the network from a more abstract level. These constructors viewed a node in the network as an instantiated generic or access node (with bound instantiation parameters) which had zero, one or two input tuples and one output tuple. Two operations were described which connected and disconnected nodes together at the tuple level. At that point a third level of abstraction, that of structures, was introduced. A structure is a portion of a network with certain constraints on the manner in which the components of that structure could be connected. Structures could be instantiated similarly to generic operator nodes. The defstruct operation viewed a node in the network as an operator which took one or two input tuples consisting of named domains and which provided an output tuple also consisting of named domains, with a convention of connecting domains of similar names. The final level, that of relational queries, is discussed below.

8.5.2. Extensions

To be relevant to the thesis, this approach must allow extension to a general monitoring system supporting the relational model. This chapter will illustrate a few ways that the system can be extended into a general system. Once the initial system has been implemented, these ideas can be investigated further.

8.5.2.1. Static Relations

Static relations can be stored by providing a *Store* operator node which would store incoming tuples in an external relational database in a relations specified by one of its instantiation parameters. Queries on stored relations could be handled with a *Retrieve* access node which would generate the tuples of a specified relation in chronological order.

8.5.2.2. Calculus-based Query Languages

The mapping of Quel statements into algebraic operator trees is well-understood, although temporal operators may introduce some complexity. The defstruct operation provides an example of compiling an algebraic specification into a network. It should be possible to extend this mechanism to handle Quel-like statements.

8.5.2.3. Efficiency

One of the motivations for an initial implementation is to determine how much of a problem efficiency will be. If it is indeed a problem, there are several possible areas for improvement. The queries presented to defstruct are in an appropriate format for transformations to improve efficiency. Such transformations sometimes provide exponential speed-ups in execution time. The algorithms within the operator nodes is another such area.

8.5.2.4. Incomplete Information

This topic was discussed briefly in my thesis proposal. The approach was to divide periods into portions which are known and portions which are possibly invalid. Each portion would be represented by a tuple with an additional implicit domain (probably a boolean value) specifying how definite the tuple was. The access nodes would initialize this domain and the operator nodes would use the domain in their update algorithms.

8.5.3. Unresolved Issues

There are many additional issues which must be resolved during the implementation of this design. A few of the more important ones are listed below:

Duplicate tuples	Should they be removed by each node, or should there be a special operator node whose only job it is to remove them? Should this node be automatically linked in by the defstruct operation? Which operator and access nodes are sensitive to duplicate tuples?
Monitoring errors	Should there also be *compensation links* between various access nodes which describe the impact one access mechanism has on the validity of the information returned by a second access mechanism?

Triggering	Some nodes (such as ApplyOp) should trigger on the leading edge of a tuple (i.e., when it becomes true), other nodes (such as Duration) should trigger on the trailing edge (when it becomes false), and a third set of nodes (such as Join) should trigger on both edges. How is this variation handled?
Initialization	What sorts of activity should take place when a node is created or linked into the network? For instance, operator nodes must initialize their internal data structures and access nodes should establish contact with the access module on Cm*.
Naming within defstruct	How are access and operator node domain names known to defstruct? How does defstruct handle domains which may or may not be input domains (such as the Function domain of ModuleProcs)? How should constants be handled?
Type checking	Should the link and defstruct operations do type checking on the pins that are connected?
Sampling	How does the monitor determine when to start sampling and the frequency to sample?

Appendix IV Representation Issues

IV.1. Tuples

Tuples are represented by a list of domains. Each domain has a type-dependent structure. The domains are referenced by index: domain 0 is the implicit time domain, and domain 1 through domain n are the explicit domains. The time domain is a list containing the begin time and the end time using a global clock.

IV.2. Domain Types

Type	Representation
INTEGER	Integer
BOOLEAN	Either the atom True or the atom Nil
CAPA	Integer index into access module's capa list (256 entries)
ENUMERATED	Symbolic value created by access node
STRING	Lisp string
TEMPORAL	List containing the slope and the intercept (initial value); the range is time

An example of an ENUMERATED domain is the type domain for the CapaInfo access node, which can have the symbolic value *BasicObject*, *CapabilityMailbox*, etc., each of which is a symbolic value within Lisp. An example of a TEMPORAL domain is the duration domain of the Duration operator node, which consists simply of a slope equal to 1 and an intercept equal to 0.

IV.3. Arithmetic Operations on Temporal Domains

Operation	Result
Temporal [+ / -] Constant	intercept' := intercept [+ / -] constant
Constant [+ / -] Temporal	intercept' := constant [+ / -] intercept
Temporal [+ / -] Temporal	intercept' := intercept_1 [+ / -] intercept_2; slope' := slope_1 [+ / -] slope_2

Temporal * Constant	intercept' := intercept * constant; slope' := slope * constant
Constant * Temporal	intercept' := constant * intercept; slope' := constant * slope
Temporal / Constant	slope' := slope / constant
Temporal / Temporal	intercept' := $slope_1$ / $slope_2$; slope := 0 (i.e., no longer a temporal domain)
all other operations	illegal

9. A General Monitoring Mechanism

Richard Snodgrass

9.1. Introduction

This document describes a very general monitoring mechanism for multiprocessors which was designed by Bob Chansler, Ivor Durham, Anita Jones, Zary Segall, and Richard Snodgrass. The fundamental approach was to design a single mechanism which could handle the majority of monitoring tasks yet be relatively simple to implement.

The intent of this document is to review the basic design, enumerate the various issues which must be considered when implementing the mechanism and then present a system-independent specification of a minimal (stage 1) monitoring mechanism. A brief discussion of the implementations of the stage 1 mechanism under two operating systems, StarOS and Medusa, completes the document.

The mechanism, like the operating systems it will be implemented on initially, is object-oriented. Monitoring actions are associated with the objects supported by the operating system and with the active entities which perform operations on these objects. The mechanism can be configured along several dimensions, allowing the user to specify quite precisely the aspects (s)he is interested in. In addition to being very flexible, the mechanism presented here is simple and efficient. The processing of monitoring data is divided into two portions, one synchronous with the occurrences the user is monitoring, and one asynchronous, with some coordination between the two. The synchronous portion is designed to be as efficient as possible, with the asynchronous portion being less critical in a multiprocessor environment.

This mechanism borrows heavily from the Hydra Kernel Tracer, which turned out to be very useful for monitoring both the operation system and user processes.

9.2. The Basic Design

The monitoring mechanism developed here records information concerning events, which are simply occurrences the user is interested in. Given this general definition of events, the mechanism can be used to support both tracing and sampling. There are several components to the mechanism:

- *events*, which are actions performed by an *initiator* to an object
- *event types*, which partition events into useful categories
- *sensors*, which detect that an event has occurred

- a *storage mechanism* for event information
- a *receptacle* for holding event information
- a *notification mechanism* for performing auxiliary actions when event information is stored
- a *sensor enable mechanism* for turning on and off the sensing of events
- a *storage enable mechanism* for turning on and off the storage of event information
- a *notification enable mechanism*
- a mapping between sensors and event types
- a mapping between event types, initiators, and receptacles
- a mapping between event types, objects, and receptacles
- a *compensation mechanism* to aid in calculating the effect the monitoring action had on the data in the collected information

In order to achieve maximum generality, it is useful to consider each of these components as a separate entity. This approach will be taken here, with the eventual implementation enforcing certain assumptions concerning these components.

9.2.1. EventSets

For each type of object to be monitored, there is an *eventset* of event types concerning objects of that type. The following is a list of possible eventsets:

- Process EventSet
- Task Force EventSet
- Multiplexor EventSet
- Memory Management Table EventSet
- Image Slice EventSet
- Module EventSet

Each eventset contains several sources of monitoring information for the event types contained in the set. The events and information sources for the Process EventSet include:

- Assign event - scheduler

- Block event - multiplexor
- Create event - process creator
- Preempt event - multiplexor
- Wakeup event - multiplexor

9.2.2. Receptacles

Receptacles are used to store the information associated with an event. Each Receptacle contains

- internal state, and
- a circular queue of event records.

Intuitively, an eventset is a collection of sources of monitoring information concerning the object type the eventset is associated with, and a receptacle is a container for this information. Receptacles contain an enable boolean in their internal state which specifies whether event records are to be stored in the receptacle.

The implementation determines the *grain* of events which can be monitored. Clearly, if the monitoring mechanism is to not greatly perturb the system, the grain of events to be monitored must be significantly larger than the combined storage and notification mechanisms.

9.2.3. Event Records

Event records contain

- event information,
- an optional timestamp, and
- optional monitoring error compensation information.

9.2.4. The InitiateMonitoring Operation

To associate an event type, an object, and a receptacle, the

```
    InitiateMonitoring (Object, EventType, Receptacle)
```

operation is provided by all eventsets. There is also an analogous TerminateMonitoring operation. As an example which will be referred to throughout this document, the

```
    InitiateMonitoring (AProcess, Block, ThisReceptacle)
```

operation provided by the ProcessEventSet connects ThisReceptacle to the Block event concerning

AProcess. When AProcess blocks, an event record will be stored in ThisReceptacle. In this case, the initiator of the event is the multiplexor and the sensor that detected the event is contained there. The sensor of another event concerning processes could be located in a user process which sampled a process and then generated an event containing the information just gathered. If this event is designated a *query* event, then the operation

```
    InitiateMonitoring (SecondProcess, Query, AnotherReceptacle)
```

could also be performed. When the user process makes a query on SecondProcess, the information is recorded in AnotherReceptacle.

When an event is generated on a particular object (say the block event on AProcess), the event mechanism first determines whether the sensor detecting the event is enabled (see section 9.2.5). If not, then the event is considered to have not been detected (there is a philosophical issue as to whether the event has *actually* been detected anyway). If the sensor is enabled, the event has definitely been detected, and the object is checked to ensure that there is a receptacle associated with that object and that event. If so, the receptacle is checked to see if it is *enabled*. No event records are stored in a disabled receptacle. After the information is stored, a *notification* as specified in the receptacle is performed.

9.2.5. Sensors

A *sensor* is simply a mechanism for determining that an event has occurred, collecting the relevant information concerning that event, and storing that information into the appropriate receptacle. Thus, every sensor generates event records for events of a particular event type. A sensor can be enabled or disabled just as a receptacle can. However, if a sensor is disabled, no information will ever be stored in receptacles associated with that sensor. Thus, if the block event sensor is disabled, no information will be stored in ThisReceptacle regardless of whether ThisReceptacle is enabled.

By enabling or disabling a given receptacle, the monitoring data can be filtered on an object-by-object basis. A similar, though orthogonal, filtering occurs when sensors are enabled and disabled. For instance, disabling ThisReceptacle causes data concerning the blocking of AProcess (on <u>all</u> process*ors*) to be filtered out. Disabling the block sensor within a particular multiplexor causes only blocks on that processor (concerning <u>all</u> processes) to be filtered out.

Each eventset supports the sensor-event association and the object-event type-receptacle association. Thus it may be possible for an eventset to automatically enable of disable various sensors as receptacles were added or removed from objects. From a different point of view, the eventset could reject InitiateMonitoring requests if the sensors are already disabled. The eventset might also reject *all*

InitiateMonitoring operations on the Block event if the current system does not contain a sensor for this event. An arbitrary amount of intelligence could be embedded in the eventset's InitiateMonitoring and TerminateMonitoring operations. However, the minimal functionality would be easy to implement.

9.2.6. Notification

The notification mechanism performs auxiliary actions when an event record is stored in a receptacle. These actions *notify* other entities (esp. the monitoring process(es)) that the event record has been stored or that a condition (such a queue overflow in the receptacle) occurred. This mechanism is necessitated by the separation of the sensors (and the associated storage mechanism), the receptacles where the event records are stored, and the monitoring process, which removes the event records from the receptacle for further analysis. The decoupling has several advantages, including efficiency of the monitoring portion which is synchronous with the event (e.g., the storage mechanism) and the filtering capabilities mentioned in section 9.2.5. The primary disadvantage is that the occurrence of the event can be separated from the detection of the event *by the monitor* by an arbitrarily long period of time, depending on how often the monitoring process examines each receptacle.

The notification mechanism is included to inform the monitoring process that either an event record has been stored or that a particular condition concerning the state of the receptacle has been satisfied. This mechanism should transfer minimal information, thus requiring the monitoring process to examine the receptacle for a more detailed view of the condition causing the notification.

9.2.7. Summary

A number of actions must be taken in order to collect monitoring information. First, one or more receptacles must be created, and the notification mechanism must be configured for each receptacle. Then the InitiateMonitoring operation is performed for each object-event type pair for which monitoring is desired. At this point, the receptacles start filling up with event records. It is the responsibility of the monitoring process to ensure that the event records are removed from each receptacle before the information is overwritten. The notification mechanism discussed in the next chapter is designed to aid in achieving this loose synchronization.

9.3. The Basic Implementation

There are two levels to the implementation of this mechanism. The lower level (to be specified shortly) consists of the receptacle and receptacleset data types, and the ReceptacleStore operation. The upper level consists of the eventsets and the InitiateMonitoring and TerminateMonitoring operations.

9.3.1. ReceptacleSets

The mechanism allows a receptacle to be associated with a particular event type and a particular object. This association is implemented using the *receptacleset* data type, which is simply a set of receptacles. Each object to be monitored is associated with a receptacleset. Receptaclesets can be associated with any object which has associated operations written in software, such as

- processes
- system tables
- abstract data objects
- StarOS modules
- task forces
- file control blocks

If the storage and notification mechanisms are implemented in software (as is envisioned), then the objects which cannot be associated with receptaclesets are the ones supported by firmware:

- StarOS mailboxes
- StarOS carriers
- StarOS basic objects
- StarOS deques
- Medusa pipes
- Medusa descriptor lists
- Medusa pages
- directories

Note that some operations on these object types are implemented in software, so that it may still be useful to associate receptaclesets with instances of them.

9.3.2. The ReceptacleStore Operation

Information is stored in a receptacle by the *ReceptacleStore* operation which takes the following parameters:

- a receptacleset

- a sensor identifier
- event information

The ReceptacleStore operation first ensures that the sensor is enabled, and if so, that the particular receptacle is enabled. If these conditions are satisfied, then the event information is stored in the *receptacle* and an optional *notification* is performed.

This mechanism is amenable to both tracing and sampling. Tracing is accomplished by embedding ReceptacleStore operations in the code for the modules of the system. Whenever this code is executed, the event would be recorded in the relevant receptacle.

Sampling is accomplished by handing over the object to be sampled to a process designed to sample the indicated object type. Periodically this process will examine the object and then execute one or more ReceptacleStore operations.

9.4. Implementation Issues

Given this framework, there are still many design issues to be resolved. The remainder of this document will outline the various options implied by these issues.

9.4.1. Internal State of a Receptacle

The internal state of a receptacle includes one or more of the following components:

- start pointer -- refers to the beginning of the circular queue (where event records are removed by the monitoring process)
- end pointer -- refers to the end of the circular queue (where event records are added)
- cycle counter -- the number of times the end pointer has been reset to the top of the queue buffer
- cumulative counter -- only necessary if event records are of variable length
- notification mailbox
- notification receptacle
- operation register
- semaphore -- for coordinating access to the receptacle
- free record limit -- used in one of the notification predicates

9.4.2. Notification

There are two aspects to the notification issue: how and when. Possible *notification predicates* are

- when an event record is stored in the receptacle;
- when the number of free records in the queue goes below the free record limit (includes buffer-full and overwrite conditions as special cases);
- when the counter reaches a specified value;
- when the cycle counter is incremented; or
- when some other condition on the internal state is satisfied.

Possible *notification actions* are

- sending a data message to the notification mailbox indicating the current cycle count, the offset into the queue, and perhaps a timestamp;
- sending a capability message to the notification mailbox containing the receptacle which the notification concerns;
- setting or clearing enable bits in the operation register; or
- generating another event record on the notification receptacle.

9.4.3. Operation Register

The operation register specifies the action to be taken when a ReceptacleStore operation is executed. This register is a collection of switches which can be set either by a *ReceptacleSwitch* operation or by a notification. These switches enable

- the store operation itself,
- various notification actions,
- various notification predicates,
- various combinations of notification predicates and actions
- timestamps in event records,
- timestamps in notifications, and/or
- other optional operations.

9.4.4. Restrictions

Certain restrictions may be necessary in order to efficiently implement this monitoring mechanism. Possible restrictions which came up in discussions were

- Receptacles cannot be shared by several receptaclesets
- Receptaclesets cannot be shared by several objects
- the notification receptacle must be in the same receptacleset
- the reserved capability slot in processes must contain a receptacleset capability, even if all the receptacles in that receptacleset are disabled

9.4.5. Compensation

Information may be stored in the sensor or the receptacle which would aid in calculating the effect the monitoring action had on the data in the collected event records. This information may include

- an event generation counter
- a ReceptacleStore fail counter
- an overhead estimation counter

It is difficult to design a compensation mechanism without a specific implementation in mind.

9.5. The Stage 1 Specification

The stage 1 monitoring mechanism is designed to provide the minimal functionality yet still allow a significant portion of the events to be monitored.

9.5.1. Receptacles

The mechanism consists of the receptacleset and receptacle data types and the ReceptacleStore operation. Each receptacle contains

- internal state, and
- a circular queue of event records.

Event records are fixed length records containing

- event information and
- an optional timestamp.

The internal state of a receptacle contains the following components:

- start pointer
- end pointer
- cycle counter
- notification destination
- operation register
- free record limit
- event record size

The operation register contains

- the store operation enable switch
- a notification predicate indicator
- a notification action indicator
- a switch enabling timestamps in event record

9.5.2. Notification

The notification predicates which are supported are

- an event record is stored in a receptacle
- the number of free records (as indicated by the difference between the end and start pointers) falls below the free record limit

The supported notification actions are

- setting an indicator which the monitoring process(es) can test very quickly
- clearing the store enable bit in the operation register

All the restrictions listed in section 9.4.4 are acceptable.

9.5.3. The ReceptacleStore Operation

The ReceptacleStore operation will take the following parameters:

- a receptacleset,
- a sensor identifier, and

- event information.

The number of words of event information is assumed to be equal to the event record size contained in the receptacle. The basic algorithm for the operation is

```
Is the receptacleset capability nil?
     Yes - exit
Index into the receptacleset to get the receptacle
Is the receptacle nil?
     Yes - exit
Is the receptacle enabled for stores?
     No - exit
Are timestamps in event records enabled?
     Yes - collect timestamp
Store event record
Is notification predicate satisfied?
     Yes - perform notification
```

Sensors and events are separate entities. The StoreReceptacle operation is be handed a sensor identifier, which it maps into an index into a receptacleset. The sensor is disabled by disassociating the sensor identification from an event. The eventsets are responsible for maintaining the mapping from sensors to events. This mapping is modified by the InitiateMonitoring and TerminateMonitoring operations.

9.6. The StarOS Implementation of Stage 1

The StarOS group has agreed to implement the stage 1 mechanism in its entirety. This chapter will give an overview of the correspondence between the stage 1 specification and the StarOS implementation of that specification.

9.6.1. ReceptacleSets

The following objects will have a predefined receptacleset capability slot:

- processes
- modules
- task forces

Receptacles per se will not exist; instead, the receptacles will be merged into the *receptacleset*. The operation registers will form an array of operation registers, and the individual queues will be combined into a single queue. This arrangement has at least three advantages:

- the proliferation of small objects (receptacles) will be avoided;

- internal fragmentation due to the presence of many small buffers will be eliminated; and
- the monitoring process will be able to manage the various queues more efficiently because they will be larger and fewer in number.

The primary disadvantages are

- receptacles cannot be shared by several receptaclesets, and
- the event type must be stored in the event record.

9.6.2. The ReceptacleStore Operation

The ReceptacleStore operation will be implemented as a procedure call in software. Hence the grain of events which can be monitored is at least an order of magnitude greater than a procedure call.

9.7. The Medusa Implementation of Stage 1

There are two possible Medusa implementations which do not involve extensive Kmap modifications. Both implementations merge receptacles into the receptacleset.

The first approach splits the receptacleset into a data structure which is stored in the object being monitored and a conventional Medusa pipe. The pipe serves as the circular buffer and the notification mechanism. The data structure consists of a capability to the pipe and the event operation registers. The monitoring process is responsible for retrieving the event records from the pipe.

The second approach involves creating a new Medusa object which is equivalent to a page object except that

- there could be several capabilities to the same object, residing in different task forces if necessary, and
- there would be less than 2K words of data, due to the presence of back pointers within the object.

Using this new object type, an implementation similar to that of StarOS could be done.

The advantages of the first approach are

- no new object types are necessary, hence no microcode modifications are involved;
- variable length event records incur no additional overhead;
- the synchronous portion of the ReceptacleStore operation is very fast; and

- the circular buffer (i.e., the pipe) can be shared.

The advantages of the second approach are

- it is more consistent with the StarOS implementation;
- it allows sharing of receptaclesets;
- recent information is retained in preference to old information if the circular buffer is full; and
- flow control (using the free record limit) is feasible.

10. Description File Specifications

Richard Snodgrass

10.1. Introduction

During the initial development of the low level event collection mechanism for the Cm* monitoring system, it became apparent that there were several procedural difficulties in the placement of sensors in the StarOS and Medusa operating systems. One difficulty was the that the Sensor macro (which stores the event records) was becoming quite cumbersome. One design required twelve parameters for a typical sensor involving three domains! Since the sensors were to be placed in critical portions of the operating system, there was little room for error in the specification of these parameters A second problem was the assignment of event numbers; an incorrect event number in a Sensor macro would result in the absence of event records of that type--a situation that might be difficult to detect by the user interacting with SIMON. Two sensors with the same event number would cause havoc within SIMON. A third problem is maintaining consistency between SIMON's view of the world and the world as it actually is. This is especially true during the early development of the monitor, when the collection of sensors inside the operating system, and the various attributes of those sensors, is changing frequently.

Finally, all these problems are exacerbated by the sheer number of sensors: I can easily envision several hundred sensors in both StarOS and Medusa when the monitoring system has been fully developed. The task of ensuring that all of these sensors, which are distributed among many source files, are correct and consistent, both within each other and with SIMON, is unmanageable if it remains a manual one.

The solution, described in this document, is to create a database, called the *sensor description file (SDF)*, which contains information on the sensors defined in a given taskforce. As work progressed on designing the sensor description file, it became apparent that the syntax (as well as the programs processing that syntax) were general enough so that databases could be designed to contain information other than that related to the sensors. In fact, the *description files* will be used to communicate *all* static information to SIMON from other programming environment tools.

This document details the syntax and semantics of the various description files. Each chapter describes a particular use and format for the description file; a table can be found at the end of the chapter summarizing the file format. Both user-level and implementation-level information is included, usually in separate sections. This chapter provides an overview of the syntax followed by all the description files, and describes the processing of these files.

10.1.1. Syntax and Definitions

A description file consists of a set of *objects* partitioned into *classes*. Each object is associated with a set of class-dependent *attributes*. There can be one or more *values* for each attribute, and some attributes can have objects as values. To describe a particular format, one must specify the object types, the attributes for each type, and the allowable values for each attribute.

The syntax follows the above description quite closely:

```
<description file>      ::= <objects>

<objects>               ::= <object> | <object> <objects>
<object>                ::= ( <class> <attribute list> )

<attribute list>        ::= <attribute> | <attribute> <attribute list>
<attribute>             ::= ( <attribute name> <attribute values> )

<attribute values>      ::= <attribute value> | <attribute value> <attribute values>
<attribute value>       ::= <object> | <object name> | <atom> | <integer> | <string> | <list>

<class>                 ::= <atom>
<objectname>            ::= <atom>
<attributename>         ::= <atom>
```

<integer> can be positive or negative; <atom> are converted to uppercase.

As an example, the following description file has one object with two attributes:

```
(oneclass (name abc)
          (objectattribute 3)
)
```

10.1.2. The Description File Preprocessor

The *description file preprocessor* (DFPRE) reads in a description, performs syntax and semantic checking, and outputs one or more files containing information derived from the input file. The syntax is specified in another description file called a *description format file (FDF)* (see chapter 10.3 for details). The semantics is contained in a function inside DFPRE which must be provided for each format. This function performs various checks on the information in the file and derives new information to be placed in one of the output files.

Once a description file has been processed, the information can then be output in a number of ways, depending on the requirements of the format. All formats can be output as a *Simon description*. This file is a PDP-11 assembly language file which is assembled and placed in a taskforce to be loaded onto Cm*. Once the information is in the taskforce in this special format, it can be easily and efficiently sent over to SIMON by the resident monitor (see section 10.3.7).

Other object files may be produced by DFPRE. These files are dependent on the individual formats and will be described in the relevant chapter.

10.2. Sensor Description File (SDF)

There are actually two sensor description file formats, one for StarOS and one for Medusa. However, the formats are similar, so they will be discussed together. All unqualified statements apply to both Medusa and StarOS. The **Name** attribute is required for all objects. The **Documentation** (or **Doc**) attribute allows comments to be given. Note that parentheses must be matched in all attributes, including **Documentation** and **Doc**.

10.2.1. TaskForce

An SDF describes the sensors defined in a given taskforce. Since StarOS is itself a taskforce, an SDF describes all the sensors within the operating system. Medusa requires several SDF's, one for each utility.

There are only a few user-supplied attributes for TaskForces. The **SimonFileName** specifies the name of the file where DFPRE will place the Simon description (see section 10.3.7). The **Version** attribute is for documentation purposes.

10.2.2. ObjectType

This class describes the objects which can be monitored by sensors defined in this SDF. Objects monitored in StarOS must reserve one word and one capability slot for monitoring information; their locations are defined by **WordOffset** and **RSOffset**. Medusa objects must reserve 17 words starting at the field named by **ExternSensOffset**.

10.2.3. SensorProcess

The **Inline** attribute specifies whether inline code for sensors is to be used at all. This attribute, if false, overrides the **SpaceTimeRatio** attribute in events. The **RequireFileName** specifies where the sensor macro definitions are to be placed. The other attributes apply on for StarOS SDFs. The **FunctionName** and **ModuleName** attributes will be used in conjunction with the naming description file (see chapter NDF); until that interface is working the **FunctionNumber** will be used. If the process contains any internal events (see below), then the **RSOffset** and **WordOffset** attributes must be specified, analogously to ObjectTypes. The **ClockPage** attribute specifies the address, if any, the clock may be found.

10.2.4. Event

The **Location** attribute identifies the sensorprocess containing the sensor for this event. The **Object** attribute specifies the objecttype this event refers to. The **Timestamp** attribute specifies whether timestamps are to be included in the event record. Note that SIMON uses the timestamp to identify the sensor and the object referred to in the event record. The **WaitTime** attribute specifies the number of microseconds to wait for space to be made available in the receptacle for an event record. A WaitTime of 0 indicates no waiting; a WaitTime of -1 indicates infinite waiting.

The **SpaceTimeRatio** attribute indicates the relative tradeoff desired between space and time efficiency in the sensor. A high spacetimeratio (near 100) specifies a sensor which optimizes space as much as possible; analogously, a low spacetimeratio (near 0) optimizes the time efficiency. The **InLine** attribute is equivalent to a low spacetimeratio if true and a high spacetimeratio otherwise.

Several attributes relate to event enabling. If the **AssumeEnabled** attribute is true, then the event is always enabled. This attribute is useful for message- and invocation-sampled events (see below). If the **CheckOneTime** attribute is true, then the onetime bit will be queried after the event record is stored to determine whether to turn off the enable bit. If the checkonetime attribute is false or not specified, then the **AssumeOneTime** attribute must be specified. If true, then the enable bit will always be turned off after the event record is stored; otherwise, the enable bit will not be touched.

The **MinorType** attribute specifies the event type. There are five types of events: object-traced, sensor-traced*, receptacle-sampled*, message-sampled, and invocation-sampled. The event types designated by (*) are **Internal** (a generated attribute, see below), indicating that the receptacleset contained in the sensorprocess, rather than in the object, is to be used.

A traced event record is written *when the event occurs*. The traced events are enabled by setting the appropriate bits in the appropriate receptacleset. An object-traced event uses the receptacleset associated with the *object* to determine the various aspects of the event record: whether it is to be stored at all, where it is to be stored, etc. A sensor-traced event uses the receptacleset associated with the *sensor* for this information. When a sensor-traced event occurs (an example is the reading of a file), the process sensing the event (in this case, a filesystem process) will first check the store enable bit in its internal receptacleset. If this bit is set, then the process will store the event record. As far as the implementor of the process is concerned, the only change in the source code will be a single Sensor macro call. When an object-traced event occurs (the reading of a file might be both a sensor and an object-traced event), the same sequence is executed, except that the object's (in this case, the file control block object) receptacleset is used.

A sampled event record is written at the request of SIMON. Sampled events require more cooperation from the implementor of the process sensing the event. Sensors for receptacle-sampled events still refer to the enable bits located in the sensor process's internal receptacleset. However, these bits are not checked when an operation is performed; instead, the process must periodically query these bits. If the bit for an receptacle-sampled event is set, then the sensor process must generate the event records *at that time*. In the usual case, the onetime bit will also be set, resulting in the clearing of the enable bit after the event records are generated. An example of a receptacle-sampled event is an event in the process which generates an event record specifying the number of free words in a data structure within the process. To determine this value, SIMON tells the resident monitor to enable this particular event; the event record is generated the next time the enable bit is queried by the process (hopefully soon after the bit is set), and the enable bit is cleared so that no more event records will be generated.

The last two types of events, message and *invocation*-sampled events, requires even more cooperation. Whenever SIMON desires an event record concerning a *particular object (and/or process or module)*, a SAMPLE command will be sent to the resident monitor, which will either send a message containing the object to the indicated process, which must then generate the requested event record(s), or perform a function invocation on the indicated module, which must also generate the requested event record(s). In either case, it is the implementor's responsibility to see that the message or invocation is received and the event records are produced.

Although the minortype, onetime, and assumeonetime attributes are specified independently, most of the time the values will follow these patterns:

(minortype object-trace)	(checkonetime t)	
(minortype sensor-trace)	(checkonetime t)	
(minortype receptacle-sample)	(checkonetime nil)	(assumeonetime t)
(minortype message-sample)	(checkonetime nil)	(assumeonetime nil)
(minortype invocation-sample)	(checkonetime nil)	(assumeonetime nil)

The **DeclareObject** attribute causes the sensor to declare the object to the resident monitor so that the resident monitor can later associate a receptacleset with this object. This attribute will probably be set for create (sensor-traced) or receptacle-sampled events, and are relevant only for StarOS sensors. The **Domains** attribute lists the domains included in this event.

10.2.5. Domain

Since domains have attributes of their own, they are treated as separate objects. Domains are ordered within events. The **Type** attribute specifies the format of the domain within the event record.

10.2.6. Generated Attributes

DFPRE generates attributes to be used by SIMON and to be used in producing require files. The **Index** attribute for domains indicates the relative ordering of the domains within the event, with the first domain having an index of 1. The **EnableBitStatus** attribute is derived from the **AssumeEnabled**, **AssumeOneTime**, and **CheckOneTime** attributes. The **Internal** attribute is derived from the **MinorType** attribute. The **EnableIndex** attribute specifies the bit position in the receptacleset associated with this event, and is also used as the event number.

Several attributes are generated for both object types and sensor processes, describing aggregates over the events defined for those objects. The number of enable bits in the external receptacleset for an object type (and the internal receptacleset for a sensor process) is recorded in the **NumEnableBits** attribute. The **Events** attribute contains all the events associated with the object type or sensor process.

If one of the events associated with a sensorprocess is internal, then the **InternalEvent** attribute is true. The data and time the SDF was processed by DFPRE is recorded in the **Minute**, **Hour**, **Day**, **Month**, and **Year** attributes; the name of the SDF appears in the **SDFileName** attribute.

10.2.7. Require Files

DFPRE produces several files from an SDF: (1) a Simon description to be loaded with the taskforce and sent to SIMON by the resident monitor; (2) a require file containing definitions for each sensorprocess (in the case of a StarOS SDF) or the taskforce (for a Medusa SDF); and (3) a require file for each object type. The goal is to imbed much of the detail of the event collection mechanism in these require files, so that few modifications are necessary to the source code of the processes containing the sensors.

10.2.7.1. Sensor Process Require Files

The sensor process require file has the following components (*Name* is replaced by the name of the sensor process):

- MakeInternalRS macro (if internal events are defined for this process--expands to a call to the MakeReceptacleSet macro);
- *require statements for the receptacleset definitions and for each object type referenced by the* process;
- *Name*sensor macro (for each event located in this sensor process--expands to a call to the Sensor macro);
- NumberOfEvents definition;
- MaxRecordSize definition.

10.2.7.2. Object Type Require Files

The require file for each object type contains the following components (*Name* is replaced by the name of the object type):

- Make*Name*RS macro (expands to a call to the MakeReceptacleSet macro);
- NumberOf*Name*Events definition.

10.2.8. Summary

Class[8]	Attribute	Value[9]	Notes[10]	Description
Class	Doc	Anything		
Class	Documentation			
Class	Name	Atom	O,R	
Domain	Index	Integer	G,O	
Domain	MaxSize	Integer[11]	?	

[8]If *Class*, then attribute applies to all classes.

[9]Either a defined type (anything, atom, boolean, or integer), a class name, or a special type, whose values are listed in a footnote.

[10]G = generated by DFPRE, M = only for Medusa, O = output to Simon Description, R = Required, S = only for StarOS, T = if in a Medusa SDF, then specified in the Taskforce, but generated for the sensorprocess, X = will eventually be removed, ? = not documented.

[11]Only if the Type is string.

Domain	Type	DomainType[12]	O,R
Event	AssumeEnabled	Boolean	
Event	AssumeOneTime	Boolean	
Event	CheckOnetime	Boolean	
Event	DeclareObject	Boolean	S
Event	Domains	Domain	
Event	EnableBitStatus	Atom[13]	G
Event	EnableIndex	Integer	G,O
Event	IndexCap	Integer	G
Event	Inline	Boolean	
Event	Internal	Boolean	G,O
Event	Location	SensorProcess	R
Event	MinorType	MType[14]	O,R
Event	Object	ObjectType	O
Event	SpaceTimeRatio	Integer	
Event	StartCap	Integer	G
Event	TimeStamp	Boolean	S
Event	WaitTime	Integer	
ObjectType	Events	Event	G
ObjectType	ExternSensOffset	Atom	M,X
ObjectType	MaxEventSize	Integer	G,?
ObjectType	MinEventSize	Integer	?
ObjectType	NumEnableBits	Integer	G
ObjectType	RequireFileName	Atom	S
ObjectType	RSSlot	Anything[15]	S,R,X
ObjectType	WordOffset	Anything	S,R,X
SensorProcess	ClockPage	Anything	S
SensorProcess	Events	Event	G
SensorProcess	FunctionName	Atom	O,S
SensorProcess	FunctionNumber	Integer	O,S
SensorProcess	Inline	Boolean	T
SensorProcess	InternalEvent	Boolean	G
SensorProcess	MaxEventSize	Integer	G,?
SensorProcess	MinEventSize	Integer	T,?
SensorProcess	ModuleName	Atom	O,S
SensorProcess	NumEnableBits	Integer	G
SensorProcess	Objects	ObjectType	G
SensorProcess	RequireFileName	Atom	T

[12]One of DoubleInteger, Integer, or String.

[13]One of CheckOneTime, ClearStoreEnable, DontTouchStoreEnable, AssumeEnabled.

[14]One of ObjectTraced, SensorTraced, ReceptacleSampled, MessageSampled, or InvocationSampled.

[15]Usually an integer or an atom.

SensorProcess	RSSlot	Anything	R,S,X
TaskForce	Day	Integer	G,O
TaskForce	Hour	Integer	G,O
TaskForce	Inline	Boolean	M
TaskForce	SimonFileName	Atom	
TaskForce	MinEventSize	Integer	M,?
TaskForce	Minute	Integer	G,O
TaskForce	Month	Integer	G,O
TaskForce	PipeName	Atom	M
TaskForce	RequireFileName	Atom	M
TaskForce	SDFileName	Atom	G,O
TaskForce	SystemName	Atom	G,O,S,?
TaskForce	Version	Integer	O
TaskForce	Year	Integer	G,O

10.3. Format Description File (FDF)

Each description file format is associated with a set of classes, a set of attributes for each class, and a set of values for each attribute. For instance, the event class can be used in a sensor description file, the **TimeStamp** attribute is defined for events, and the values True and False are acceptable for the Timestamp attribute. Rather than imbedding this format-specific information in DFPRE, it is placed in a file called a *format description file* (FDF). Using a FDF has several advantages: the various classes, attributes, and values for a format can be modified easily by the format designer, the structure of DFPRE is less complex, one version of DFPRE works for several formats, and the same routines in DFPRE which process the input description file (the SDF, etc.) can be used to initially process the format description file, since the FDF is in the same syntax as the other description files.

The classes in the FDF specify the tasks to be performed when reading in the input file (the FieldTypes, Input, and Default classes), the processing of the input file (the General class), and the generation of the Simon description (the Output class). Each of these classes will be described here. This chapter also contains a description of the internal structure of DFPRE. Perhaps the best way to understand this material is to study several existing FDF's to see how the various classes are used.

10.3.1. FieldTypes

The allowed values for an attribute are divided into (potentially overlapping) *fieldtypes*. The predefined fieldtypes are

boolean	True, T, False, F
anything	A general list except nil

Lisp Predicate The most common values are *fixp* (integers) and *atom* (names).

The field types class allows the format designer to define new fieldtypes for a particular format. The attributes of the field types class are the names of the fieldtypes, and the values are the values that belonging to that fieldtype. For instance, the specification

```
(fieldtypes   (atype 1 2 3 a b c)
              (anothertype one two three a)
)
```

includes two field types (atype and anothertype); the former allows six possible values and the latter, only four. These field types can later be used as values of attributes of the Input class.

10.3.2. Input

This class specifies the typechecking to be done on the file being read in. Each object in this class describes the valid attributes for a class in the input file and the values of these attributes indicate the fieldtype of the value of the attribute when given in the input file. For instance, the specification

```
(input        (specifies oneclass)
              (objectattribute fixp)
              (asecondattribute atom)
)
```

describes a class called oneclass, which has two attributes. The first attribute, objectattribute, can have integer values. The other attribute, called asecondattribute, can have names as values.

10.3.3. Default

The default class defines values to be assumed if none are specified by the user. The syntax is similar to the input class:

```
(default      (specifies event)
              (timestamp f)
)
```

Note: This class is not implemented.

10.3.4. General

This class controls the processing of input files. The successive steps are:

- read in the appropriate FDF, with type-checking disabled;
- read in the input file, performing type checking as specified in the input class;

- add defaults if necessary as specified in the default class;
- perform any format-specific processing (by calling the **ProcNodeFunction**);
- ask for commands and generate output files (by calling the **CommandFunction**).

The two functions must be defined in DFPRE (see the source code for DFPRE for examples of ProcNodeFunction and CommandFunction). The **FileDefault** attribute applies to the input file; the **RequireFileDefault** applies to any require file generated by DFPRE.

10.3.5. Output

In the usual case, only a subset of the attributes are sent to SIMON via the Simon description. The output class specifies which attributes are to be placed in the Simon description, and the format of the values for the attributes.

10.3.6. Internal Representation

As the input file is read in, an internal representation of the objects defined in the file is constructed. This representation is based on *frames*, which are attached as properties to atoms. Frames are composed of *slots*, which are further composed of *facets*, which have *values* (note the intentional correspondence with objects, classes, attributes, and values). An atom is generated for each object defined in the input file.

10.3.7. Simon Description Format

The Simon description file consists of a list of triples, each containing two integers and either a third integer or a character string. The first integer indicates the attribute, the second, the object, and the third element is the value of this attribute for this object. Since attributes are represented by integers, there must be some way to associate an attribute with an integer. This is done using four "meta" attributes: DefineNodeAttribute (value 1), for attributes having objects as values; DefineLitAttribute(2), for attributes having either integers or strings as values; DefineAAttribute (3), for attributes having other attributes as values, and DefineNode (4), to define classes. The example given earlier:

```
(oneclass (name abc)
          (objectattribute 3)
)
```

would be described by these triples:

```
(4 {DefineNode}, 5, "OneClass")                    oneclass = 5
(5 {oneclass}, 1, "abc")                           object #1
(2 {DefineLitAttribute}, 6, "objectattribute")     objectattribute = 6
(6 {objectattribute}, 1, 3)
```

In this way, SIMON needs only to know the values for the four meta-attributes; the names of the classes and attributes are send along with the rest of the description.

10.3.8. Summary

Class	Attribute[16]	Value	Notes[17]	Description
Default	*Attribute*	*Default Value*		
Default	Required	Attribute	?	
Default	Specifies	Class		
FieldTypes	*Type*	*Allowed Values*		
General	CommandFunction	Function Name		
General	FileDefault	Filespec[18]		
General	ProcNodeFunction	Function Name		
General	RequireFileDefault	Filespec		
Input	*Attribute*	Type		
Input	Specifies	Class		
Output	*Attribute*	OutputType[19]		
Output	Specifies	Class		

As the input file is read in, the following attributes are generated within DFPRE:

Node Name[20]	Slot[21]	Facet	Value[22]	Notes[23]	Description
All Nodes	Node	Class	Class		
All Nodes	Node	ID	Integer	O	
All Nodes	Node	IsNode	True		
All Nodes	Node	Name	Atom	O	

[16] If in *italics*, then substitute a defined attribute or type.

[17] ? = not documented.

[18] A filespec refers to a (partial or full) name for a file, and may be a list or a string. If a list, it may be in one of two formats: ((dev dir) name ext) or (name ext dev dir). The usual value for this attribute is "*.dfs".

[19] One of Attribute, Boolean, *Class*, Integer, or String

[20] The frame name is identical to the name of the node.

[21] If a *class*, then substitute a defined class.

[22] A Node is a generated atom associated with a frame of the same name.

[23] O = Output to Simon.

Name	*Class*	Node	Node
ThisFile	Attributes	*Attribute*	Integer
ThisFile	*Class*	*Attribute*	*Value*
ThisFile	General	*Class*	Node
ThisFile	General	NodeList	Node
ThisFile	General	NumAttributes	Integer
ThisFile	General	ObjFileName	Atom
ThisFile	General	SDFileName	Atom

On the SIMON side, the structure is somewhat different, emphasizing access of an attribute quickly given a Descr node[24]:

Node Name[25]	Slot	Facet	Value	Notes[26]	Description
Descr	AttributeList	*AttributeType*[27]	Attribute		
Descr	Attributes	*AttributeID*	Attribute		
Descr	*Class*	*Name*	NodeID		
Descr	Class	*Class*	NameNode		
Descr	Class	*NodeID*	Class		
Descr	General	Complete	Timestamp		
Descr	General	DescrType	DescrType		
Descr	General	NumAttributes	Integer		
Descr	General	ObjectName	ObjectName		name of description object
Descr	General	TaskForce	TFNode		
Descr	Name	*NodeID*	Atom		
Descr	NodeList	*NodeID*	Node		
General	Accountant	BufferSize	Integer		
General	Accountant	TraceToggle	Integer		
General	*DescrType*	*ObjectName*	DescrNode		name of task force
General	Derived	*RelationName*	NodeName	C	update node name
General	DescNodes	*ObjectName*	DescrNode		name of description object
General	Domains	*RelationName*	DomainName	C	
General	EventClass	MaxValue	Integer		
General	EventClass	*ObjectName*	Integer		name of a task force
General	NumOfClass	*Class*	Integer		
General	OS	RMName	ObjectName		name of resident monitor
General	OS	TFName	ObjectName		name of task force

[24]If in *italics*, substitute a value of the indicated type. A *NodeID* is a small integer, as is an *AttributeID*. Nodes and Attributes are assigned ID's starting at 1 in each *Descr*. An *ObjectName* is a (large) integer. A *DescrType* is one of Hardware, Naming, or Sensor.

[25]If in *italics*, then substitute a generated name. The frame name is identical to the node name.

[26]C = Only in monitor core, * = Not implemented.

[27]One of aattributes, litattributes, nodeattributes, nodes.

General	TaskForces	TFNodes	TFNode		
General	TupleVariables	*TVName*	RelationName	C	
General	*ObjectName*	*UserName*	ObjectName		Second Obj is a component of first Obj
General	Primitive	*RelationName*	NodeName		update node name
Mod	General	Name	Atom		
Mod	General	ObjectName	ObjectName		
Mod	Process	*FunctionNumber*	ObjectName		
Name Node	*Class*	*Attribute*	*Value*		
Name Node	*Class*	Name	Atom		
Name Node	*Class*	UniqueID	Integer		uses General - NumOfClass
Name Node	Node	Class	Class		
Name Node	Node	Descr	DescrNode		
Name Node	Node	ID	NodeID		
Query	General	TupleVariables	TVName	C	
Query	General	Relations	RelationName	C	
TF	Components	*Name*	ObjectName		
TF	Components	ModulesNode	ModuleNode	*	
TF	Description	*DescrType*	DescrNode		
TF	General	EventClass	Integer		
TF	General	Name	Atom		
TF	General	ObjectName	ObjectName		
TF	General	Processes	*ObjectName*	C	

11. The Cm* / Simon Protocol Specification

P.T.Highnam and Richard Snodgrass

11.1. Introduction

This document details in a fairly system independent way the relationship between the resident monitors on Cm* and the master monitoring process SIMON on a VAX. The resident monitors are STARMON under the StarOS operating system, and MEDIC under the Medusa operating system, communicating with SIMON under the Unix operating system via the EtherNet.

We describe the EtherNet communication protocol, the packet formats, the command and data record formats, and the initialisation and naming scheme. The details of the sensors, the internal design of SIMON, and the system-dependent aspects of STARMON and MEDIC are described in separate documents and in section 11.7.1.

11.2. EtherNet protocol

The conversation is conducted using PUPs (PARC universal packets) and a protocol similar to that of EFTP (EtherNet file transfer protocol) *simulating a transmission from SIMON to the resident monitor.* The PUP types sent to the resident monitor will be **EFTPData** with the complete communication terminated by an **EFTPEnd** packet from SIMON. The resident monitor will always be sending **EFTPAck** packets containing monitoring data as an acknowledgement for each **EFTPData** packet. Nothing will be sent by the resident monitor until requested -- this applies to every transmission. Thus, SIMON always serves as the master, and the resident monitor as the slave in the protocol. One should note that the PUP types are totally arbitrary and were chosen so that the implementors could look up the values independently.

Each **EFTPData** packet will contain a sequence number, which starts at 0 and is incremented by one with each successful packet transmission. The **EFTPAck** packet will contains the sequence number of the **EFTPData** packet it is acknowledging. The resident monitor will resend the last **EFTPAck** packet if it gets a packet with an invalid sequence number. Similarly, SIMON will retransmit an **EFTPData** packet if an invalid **EFTPAck** packet is received or if a proper acknowledgement is not received in a reasonable amount of time. This, SIMON and the resident monitor should be out of synch by no more that one packet.

Each packet will be used to carry multiple commands from SIMON or multiple data records from the resident monitor (*see chapter 11.4*). *Note that the data part of a packet may range between 0 and 266 (decimal) words in size (each word is two bytes, or 16 bits).* The Pup encapsulation uses another 11 words.

There is no notion of time in the resident monitor; any time-outs will be generated solely by SIMON. When the resident monitor receives an EFTPEnd packet, it should return to its initial state (see chapter 11.6).

11.3. Object Identification

A single 21-bit entity (called the *object-id*) will identify every object known to the resident monitor. The format of this word is defined solely by the internal considerations of the monitor. If the object associated with the *object-id* is deleted, the resident monitor is required to send a NEW NAME data record (see section 11.4.3) containing a timestamp which indicates that this *object-id*, if used again in an event record which is stamped with a time after that in the corresponding NEW NAME data record, will refer to a different object (see section 11.7.1 for details on how this data record is handled).

More comments on the interpretation of the *object-id* by the resident monitor, in particular the means chosen to deal with object creation and deletion, can be found in the STARMON and MEDIC documents.

11.4. Packet Formats

11.4.1. Parameter Sizes

The following lists the sizes for the various parameter types used in this chapter:

Type	Size in bytes	Size in bits
Command	1	8
Flag	-	1
object-id	3	21
entry-id	1	8
event-number	1	7
Data Record type	1	8
Data Record length	1	8
Timestamp	4	32
Entry value	4	32

11.4.2. Command Format (Simon to Cm*)

```
<- Command -><- Flag(s) -><- Long Parameter -><- Short Parameter ->
    byte         byte         4 bytes              2 bytes
```

Each command packet will contain several commands. The last command in the sequence will be an

END command. The commands are shown below; with each is (a) a numeric code (intended to be used to identify the corresponding command in the implementation), and (b) parameters. If there is a single END command then it is in response to a LAST RECORD record from the resident monitor with a parameter value greater than zero (indicating that there are further data records waiting to be sent). The list below summarizes the commands. The parameters are in the order of *Long Parameter, Short Parameter, Flag(s).*

ADJUST OBJECT[1]	*object-id bit-position bit-value* Alter the specified enable-bit of the specified object. The *bit-position* specifies the word offset (top 12 bits) and the bit within the word (bottom 4 bits). No response required.
CHECKPOINT[2]	-- *object-type* This is a checkup on a particular object-type. This will tell SIMON that there no events on objects of this type occurring before, but unaccounted for at, a particular time after this command was received. The CHECK data record specifies this time as a timestamp. In the first version the *object-type* will be ignored, and all pipes will be check-pointed. If a CHECKPOINT command arrives before an earlier one has been completely processed, it will be ignored.
READ ENTRY[3]	-- *entry-id* There are some things wired into the resident monitor, for example, how many pipes are in use, where to find the current time, etc. This command requests the dispatch of the aforesaid value, as a *Report* data record [2 words]. This command and WRITE ENTRY have some common entries. Appendix I contains a cumulative list and description.
WRITE ENTRY[4]	*value entry-id* The complement of READ ENTRY. Appendix I contains a tentative list and description. No response required.
SAMPLE[5]	*object-id event-number* This is a request for a particular event on a specified object to be sampled and sent back to SIMON as an EVENT RECORD. The semantics for this command is system-dependent.
END[6]	This indicates the last (physically) command in this packet.

11.4.3. Data Format (Cm* to Simon)

Normally the resident monitor picks up every event it receives and puts it into an **EFTPAck** packet for transmission when it receives an **EFTPData** packet from SIMON -- such items are called EVENT RECORDS. Response to specific enquiries, immediate (READ ENTRY) or instigated (CHECKPOINT) is made using the REPORT and CHECK data records, respectively. The termination of a data packet is indicated by a LAST RECORD record.

Should a particularly unpleasant situation be discovered by the resident monitor there is provision for notification of the error to SIMON using an ERROR data record. This will occupy three words, the second word being an *error number*, and the third being extra information (if necessary). Of course this only makes sense if SIMON is responding!

```
<- Type ->< Length(words) ->< Parameters ->
  byte        byte             1 or more words each
```

The length specifies the number of words of parameters. The following summarises the types and gives an index number for each which is intended for use within the implementation.

EVENT RECORD[1]	*object-id* (of sensor) *event-number object-id* (of object) *optional fields* This record type is sent in response to a SAMPLE command and whenever an event is sent to the resident monitor by a sensor. The top byte is 0 and the *object-id* of the sensor is in the next 3 bytes. The next byte holds the *event-number* and the next three bytes, the *object-id* of the object. The optional fields follow, word-aligned.
REPORT[2]	*entry-id value*[2 words] In response to a READ ENTRY.
CHECK[3]	*timestamp*[2 words] In response to a CHECKPOINT. The timestamp is the monitor-saw-it-when time. This data record is generated after all of the event records relating to this *object-type* have been removed from internal buffers and sent to SIMON. In future versions, an *object-id* parameter may be added.
LAST RECORD[4]	*word-count*[1 word] The physically last record within a data packet. The count tells how many words of data there were waiting to be sent when this packet was sent.
ERROR[5]	*error-number*[1 word] *additional information* [2 words] Panic time. The resident monitor has discovered something amiss (see Appendix II for a list of possible errors).
NEW NAME[6]	*object-id timestamp*[2 words] The object corresponding to this *object-id* has been deleted by the operating system at *this* time. Any reference to this *object-id* in the future (i.e., with a later *timestamp*) will involve a different object. Note that SIMON might never have heard about this *object-id* before.

11.5. Simon Description Page Format

A Simon Description Page contains information which is specially formatted to allow easy transmission over the EtherNet to SIMON. There are currently two types of Simon Description Pages, both associated with task forces: SensorDescription and NamingDescription. The location of the page is of course system-dependent. These pages are produced by a preprocessor from sensor description file (SDF), or a naming description file (NDF), respectively. For information on the contents of this page, see the Sensor Description File Specification.

A Simon Description Page is formatted as a contiguous list of words (a read-only page in Medusa and a data-only basic object in StarOS) containing one or more triples. The first two elements of each triple are integers; the third is either an integer (if the first element is positive) or a string (if the first element is negative). A first element of zero indicates that there are no more triples. A string is encoded as a zero-terminated sequence of bytes. All elements are word-aligned.

The RequestSensorDescription sensor (an internal sampled event) will send the triples from the SensorDescription page of the object designated by the *object-id* over as SimonDescriptionInteger and SimonDescriptionString event records. These events are both internal traced events.

11.6. Initial Connection

When the resident monitor first comes up on Cm* (presumably when the operating system is booted), it will open the correct socket (see below) and wait for a packet to be delivered to this socket. All monitoring activity should be turned off while the resident monitor is in this initial state. The initiation of communication will be carried out by SIMON sending a **EFTPData** packet to the following socket on Cm*[28] :

Host Net: # 52 (CMU local network)
Host Number: # 222 (CML2) for Medusa, # 223 (CML5) for StarOS
Socket Number: # 135,0 (low,high)

The sequence number carried by this initial packet will be (0,0). The initial packet from SIMON will also carry the net, host and socket on which SIMON is living. These will be picked up the Cm* process and used thereafter. Note that this allows SIMON to live anywhere. The data contained in this first packet has not been specified. The resident monitor will respond, as always, with an **EFTPAck** packet. The data in this packet has also not been specified, except that it must include the *object-id* of the resident monitor.

When SIMON first makes contact with the resident monitor, it knows of three events: RequestSensorDescription, SimonDescriptionInteger, and SimonDescriptionString. The second packet sent to the resident monitor will contain exactly four commands:

[28] In the first implementation, the socket number will be fixed. Later versions may use a rendevous protocol to determine the correct socket.

```
AdjustObject    <enable SensorDescriptionInteger>

AdjustObject    <enable SensorDescriptionString>

AdjustObject    <enable RequestSensorDescription>

End
```

The third command instructs the resident monitor to send over the Sensor Description associated with the resident monitor; the RequestSensorDescription is an internal sampled event. Once SIMON receives all of these event records, it will be able to communicate fully with the resident monitor.

11.7. Interaction with Simon

11.7.1. Simon Names

The names (*object-ids*) stored in event records received by SIMON are unique except for reuse after deletion on Cm*. However, each *Simon name* must uniquely specify a Cm* object. An *epoch* (essentially a 8-bit counter) will be used to convert an *object-id* into a *Simon name*. Associated with each *object-id* will be a list of timestamp-epoch pairs. Initially, the list will contain one pair, with a timestamp of 0 and an epoch of 0. Whenever a NEW NAME data record is received, a new pair will be added to the appropriate list (determined by the *object-id* of the data record), with an epoch one greater than the last epoch and a timestamp equal to that in the data record. As event records flow over from Cm*, each 24-bit *object-id* will be converted to a 32-bit *Simon name* by concatenating the epoch which was valid when the event record was stored (usually the last epoch in the list). If no timestamp was associated with this event record, then the current epoch is used. Conversely, when a *Simon name* is to be sent to the resident monitor on Cm*, the epoch is first stripped off of the name. If the epoch is not the current epoch, then the command containing the *Simon name* must be discarded. Otherwise, the *object-id* portion of the *Simon name* is sent.

11.7.2. Event Numbers

The event number stored in an EVENT RECORD does not uniquely specify a primitive relation, since event numbers are allocated per task force and event per objecttype or sensorprocess by the SDF preprocessor (see chapter 11.5). The correspondence depends on whether the event is an internal or an external one. If the event is internal, then the actual event is determined by the *object-id* of the activity (process) containing the sensor which stored the event record. This field identifies the *sensorprocess* described by one of the SensorDescriptions sent over by the resident monitor. The events associated with this sensorprocess are searched for one with the sensorindex attribute equal to the event number.

If the event is an external one, the *object-id* is used to identify the *objecttype* (also found in a SensorDescription); the events associated with this objecttype are searched for one with the objectindex attribute equal to the event number.

Appendix V
Readable and Writable Entries

Entry [Value]	Description
# # # [?]	Maximum number of messages to be processed by a MEDIC Packet Filler before it goes back to sleep. [Read/Write]
# # # [?]	Number of commands to be performed by the MEDIC Command Server before it goes back to sleep. [Read/Write]

Appendix VI
Errors found by the Resident Monitor

Error [Value]	Parameter Description
# # # [?]	*????* In danger of losing information because buffers are getting full.
# # # [?]	*Bad Sequence Number* Out of sequence packet arrived. (Useful for looking at the EtherNet performance and for debugging.)
# # # [?]	*Bad Type* Packet with a bad type arrived. (Useful for looking at the EtherNet performance and for debugging).
# # # [?]	*Sequence Number* No END in the command packet.
# # # [?]	*object-id* Invalid *object-id* sent in a command packet.
# # # [?]	*object-id* ReceptacleSet overflow; lost information concerning this object.
# # # [?]	*object-type* ReceptacleSet overflow; lost information concerning this object type.

References

[1] R.A. Arbuckle.
Computer Analysis and Thruput Evaluation.
Computers and Automation , Jan., 1966.

[2] R.G. Arnold and E.W. Page.
A Hierarchical Restructurable Multi-Microprocessor Achitecture.
In *Proceeding of the 3rd Annual Symposium on Computer Architecture*, pages 44-45. January, 1976.

[3] J.L. Baer.
A Survey of Some Theoretical Aspects of Multiprocessing.
Computing Surveys 5(1):31-80, March, 1973.

[4] J.E.Ball, J.A. Feldman, J.R. Low, R.F. Rashid, P.D. Rovner.
RIG, Rochester's Intelligent Gateway: System Overview.
IEEE Trans. on Soft. Eng. SE-2(4), December, 1976.

[5] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes.
The ILLIAC IV computer.
IEEE Transactions on Computers C-17:746-757, August, 1968.

[6] G. M. Baudet.
The design and analysis of algorithms fòr asynchronous multiprocessors.
PhD thesis, Carnegie-Mellon University, April, 1978.

[7] *Bliss-11 Programmers Manual*
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, 1974.

[8] W. Bucholz.
A Synthetic job for measuring System Performance.
IBM Systems Journal :309-318, 1969.

[9] Michael J. Carey.
Parallel processing for power system transient simulation: a case study.
Master's thesis, Carnegie-Mellon University, December, 1980.

[10] V. Cerf.
Multiprocessors, Semaphores and a Graph Model of Computation.
Technical Report 7223, Computer Science Department, UCLA, April, 1972.

[11] D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager.
Toth, a Portable Real-Time Operating System.
Communications of the ACM 22(2):105-115, Feb., 1979.

[12] W.W. Chu, L.J. Holloway, Min-Tsung Lan, and K. Efe.
Task Allocation in Distributed Processing.
Computer, IEEE :57-69, Nov., 1980.

[13] Jaroslaw Deminet.
Experience with Multiprocessor Algorithms.
IEEE Transactions on Computers C-31(4):278 – 87, April, 1982.

[14] W.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare.
Structured Programming.
Academic Press, New York, 1972.

[15] R.C. Dugan, I. Durham, and S.N. Talukdar.
An algorithm for power system simulation by parallel processing.
In *Text of abstracts, Summer Power Meeting*. IEEE Power Engineering Society, 1979.

[16] I. Durham, R.C. Dugan, A.K. Jones, and S.N. Talukdar.
Power system simulation on a multiprocessor.
In *Text of abstracts, Summer Power Meeting*. IEEE Power Engineering Society, 1979.

[17] G. Estrin and R. Turn.
Automatic assignment of computations in a variable structure computer system.
IEEE Transactions on Electronic Computers, EC-12 :747-754, Dec., 1963.

[18] Peter Feiler.
Harpy.
In S. H. Fuller, A. K. Jones, and I. Durham (editors), *Cm* Review*, pages 57 – 64. Computer Science Department, Carnegie-Mellon University, June, 1977.

[19] Domenico Ferrari.
Workload Characterization and Selection in Computer Performance Measurement.
IEEE, Computer :18-24, July/August, 1972.

[20] S.H. Fuller and S.P. Harbison.
The C.mmp Multiprocessor.
Technical Report CMU-CS-78-148, Department of Computer Science, Carnegie-Mellon University, 1978.

[21] S.H. Fuller, J.K. Ousterhout, L. Raskin, P. Rubinfeld, P.S. Sindhu, and R.J. Swan.
Multi-microprocessors: An overview and working example.
Proceedings of the IEEE 66(2), Feb., 1978.

[22] Ilya Gertner.
Performance Evaluation of Communicating Processes.
PhD thesis, University of Rochester, 1980.

[23] K.P. Gostelow.
Flow of Control, Resource Allocation, and the Proper Termination of Programs.
PhD thesis, University of California, Los Angeles, Dec., 1971.

[24] A.N. Habermann.
An Overview of the Gandalf Project.
Carnegie-Mellon University, Computer Science, Res.Rev. 1978-1979 , 1980.

[25] C.A.R. Hoare.
Monitors: An Operating System Structuring Concept.
CACM , October, 1974.

[26] C.J. Jenny .
Process Partitioning in Distributed Systems.
Digest of Papers NTC '77 , 1977.

[27] A.K. Jones, R.J. Chansler, I. Durham, P.H. Feiler, D.A. Scelza, K. Schwans, and S.R. Vegdahl.
Programming Issues Raised by a Multiprocessor.
Proceedings of the IEEE 66(2):229-237, February, 1978.

[28] A.K. Jones and K. Schwans.
TASK Forces: Distributed Software for Solving Problems of Substantial Size.
In *Proceedings of the Fourth International Conference on Software Engineering*, pages 315-330. IEEE, September, 1979.

[29] A. K. Jones, Robert J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl.
StarOS, a multiprocessor operating system for the support of task forces.
In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 117 – 27. ACM/SIGOPS, Asilomar Conference Grounds, Pacific Grove, California, December 10 – 12, 1979.

[30] A.K. Jones, and Edward F. Gehringer, eds.
The Cm Multiprocessor Project: A Research Review.*
Technical Report, Computer Science Department, Carnegie Mellon University, July, 1980.

[31] Anita K. Jones and Edward F. Gehringer [eds.].
The Cm multiprocessor project: A research review.*
Technical Report CMU-CS-80-131, Computer Science Department, Carnegie-Mellon University, July, 1980.

[32] A.K. Jones, K. Schwans.
TASK forces: Distributed Software for solving problems of substantial size.
4th International Conference on Software Engineering :315-330, Sept., 1979.

[33] Anita K. Jones and Karsten Schwans.
Task forces: Distributed software for solving problems of substantial size.
In *Proceedings of the Fourth International Conference on Software Engineering*. ACM/SIGSOFT, Munich, September 14 – 16, 1979.

[34] A.K. Jones, R.J. Chansler, Jr., I. Durham, K. Schwans, and S.R. Vegdahl.
StarOS a Multiprocess Operating System for the support of Task Forces.
Proc. of the Seventh Symposium of O.S. Principles :117-127, Sept., 1979.

[35] E.O. Joslin.
Application Benchmarks: the key to meaningful computer evaluations.
Proc. 20th ACM Nat. Conf. :27- , 1965.

[36] Hisashi Kobayashi.
Modelling and Analysis: an introduction to system performance evaluation methodology.
Addision-Wesley Pub. Co., 1978.

[37] Henry F. Ledgard, and Michael Marcotty.
A Genealogy of Control Structures.
Communications of the ACM 18(11):629- , Nov., 1975.

[38] B. Liskov, ed.
Report on the Workshop on Fundamental Issues in Distributed Computing.
Op Sys Review 15(3), July, 1981.

[39] B. T. Lowerre.
The Harpy speech recognition system.
Technical Report, Computer Science Department, Carnegie-Mellon University, April, 1976.

[40] Henry C. Lucas, Jr.
Performance Evaluation and Monitoring.
Computing Surveys 3(3):79-91, Sept., 1971.

[41] Madhav V. Marathe.
Performance Evaluation at the hardware architecture level and the operating system kernel design level.
PhD thesis, Department of Computer Science, Carnegie Mellon University, Dec., 1977.

[42] S. R. McConnel, D. P. Siewiorek, and M. M. Tsao.
The measurement and analysis of transient errors in digital computer systems.
In *FTCS9*, pages 67 – 70. IEEE Computer Society, 1979.

[43] McGehearty, Patrick F.
Performance Evaluation of Multiprocessors under Interactive Workloads.
PhD thesis, Carnegie Mellon University, 1980.

[44] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller.
Equation of state calculations by fast computing machines.
J. Chem. Phys. 21:1087 – 92, 1953.

[45] M. Model.
Monitoring System Behavior in a Complex Computational Environment.
Technical Report CSL-79-1, Xerox PARC, January, 1979.

[46] R. Medina-Mora and P.H. Feiler.
An Incremental Programming Environment.
IEEE Trans. Software Engineering , September, 1981.

[47] David E. Morgan, Walter Banks, Dale P. Goodspeed, Richard Kolanko.
A Computer Network Monitoring System.
IEEE Transactions on Software Engineering SE-1(3), Sept., 1975.

[48] G.J. Nutt.
A Survey of Remote Monitors.
NBS Special Publication 500-42 , January, 1979.

[49] N. S. Ostlund, P. G. Hibbard, and R. A. Whiteside.
A case study in the application of a tightly-coupled multiprocessor to scientific computations.
In B. Alder, S. Fernbach, and M. Rotenberg (editors), *Parallel Computations*, . Academic Press, 1982.

[50] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu.
MEDUSA: an experiment in distributed operating system structure.
Communications of the ACM 23(2), February, 1980.

[51] J.K. Ousterhout, D.A. Scelza, and P.S. Sindhu.
Medusa: an experiment in distributed operating system structure.
Communications of the ACM 23(2):92-105, Feb., 1980.

[52] R.F. Rashid.
A Network Operating System for a Distributed Sensor Network.
Technical Report, Carnegie-Mellon University, Computer Science Department, April, 1980.
Internal memo.

[53] R.F. Rashid.
An Inter-Process Communication Facility for UNIX.
Technical Report, Computer Science Department, Carnegie-Mellon University, March, 1980.
Available as CMU-CS-80-124.

[54] Levy Raskin.
Performance Evaluation of multiple processor systems.
PhD thesis, Carnegie Mellon University, Aug., 1978.

[55] Levy Raskin.
Performance evaluation of multiple processor systems.
PhD thesis, Carnegie-Mellon University, August, 1978.
Published as technical report CMU-CS-78-141.

[56] Dennis M. Ritchie and Ken Thompson.
The UNIX time-sharing system.
Communications of the ACM 17(7):365 – 75, July, 1974.

[57] E. Rosen.
Vulnerabilities of Network Control Protocols: An Example.
ACM SIGSOFT Soft Eng Notes , January, 1981.

[58] M. Satyanarayanan.
Multiprocessors: A Comparative Study.
Prentice-Hall Inc., 1980.

[59] Karsten Schwans.
Tailoring Software for Multiple Processor Systems.
PhD thesis, Department of Computer Science, Carnegie Mellon University, Fall, 1981.

[60] R. Sedgewick.
Implementing quicksort programs.
Communications of the ACM 21(10):847 - 57, October, 1978.

[61] Z. Segall, A. Singh, R. Snodgrass, A. K. Jones, and D. P. Siewiorek.
An integrated instrumentation environment for multiprocessors.
To appear in *IEEE Transactions on Computers*.

[62] J. Shoch.
What's Different About 'Distributed Computing'?
1981.
in [38].

[63] W.L. Shope, K.L. Kashmarak, J.W. Inghram, and W.P. Decker.
System Performance Study.
Proc. SHARE 24:568-659, March, 1970.

[64] H. Shrobe.
Dependency Directed Reasoning for Complex Program Understanding.
PhD thesis, MIT, 1979.
Technical Report AI-TR-503.

[65] D. Siewiorek, M. Canepa, and S. Clark.
C.mmp: The Architecture of a Fault-Tolerant Multiprocessor.
In *Proceeding of the Seventh Annual International Conference on Fault-Tolerant Computing*, pages 37-43. June, 1977.

[66] A. Singh.
Pegasus: A Workload Generator for Multiprocessors.
Master's thesis, Carnegie-Mellon University, Department of Electrical Engineering, 1981.

[67] A.Singh and Z. Segall.
Synthetic Workload Generation for Experimentation with Multiprocessors.
, To be published 1982.

[68] T.B. Smith and A.L. Hopkins.
Architectural Description of a Fault-Tolerant Multiprocessor Engineering Prototype.
In *Digest of Papers, Eighth Annual Conference on Fault-Tolerant Computing*. June, 1978.

[69] Richard Snodgrass.
Monitoring Distributed Systems.
PhD thesis, Carnegie Mellon University, To be Published, 1982.

[70] Richard Snodgrass.
Monitoring distributed systems.
PhD thesis, Carnegie-Mellon University, 1982.
To be completed.

[71] Sreenivasan K., Kleinman A.J.
On the construction of a Representative Synthetic Workload.
CACM , 1974.

[72] M. Stonebraker, E. Wong, P. Kreps, and G. Held.
The Design and Implementation of INGRES.
ACM TODS 1(3):189-222, September, 1976.

[73] Richard J. Swan.
The switching structure and addressing architecture of an extensible multiprocessor, Cm.*
PhD thesis, Carnegie Mellon University, August, 1978.

[74] Richard J. Swan.
The switching structure and addressing architecture of an extensible multiprocessor, Cm.*
PhD thesis, Carnegie-Mellon University, August, 1978.

[75] Sarosh N. Talukdar.
On using MIMD-type multiprocessors—some performance bounds, metrics, and algorithmic issues.
In *Proceedings of the 10th Pittsburgh Modeling and Simulation Conference*, pages 1167 – 73. 1979.

[76] S. N. Talukdar, M. J. Carey, and S. S. Pyo.
Multiprocessors for power system problems.
Joho-Shori (Information Processing Society of Japan) 22(12), December, 1981.

[77] T. Teitelbaum.
The Cornell Program Synthesizer: A Syntax Directed Programming Environment.
Technical Report TR80-421, Department of Computer Science, Cornell University, May, 1980.

[78] M. M. Tsao.
A study of transient errors on Cm.*
Master's thesis, Carnegie-MellonUniversity, December, 1978.

[79] J.D. Ullman.
Principles of Database Systems.
Computer Science Press, Potomac, Maryland, 1980.

[80] L. Verlet.
Computer experiments on classical fluids. 1. Thermodynamic properties of Lennard-Jones molecules.
Physical Review 159:98 – 103, 1967.

[81] S.A. Ward.
The MuNet: A Multiprocessor Message-Passing System Architecture.
In *Prodeeding of the Seventh Texas Conference on Computing Systems*, pages 7-21. November, 1978.

[82] D.C. Wood, and E.H. Forman.
Throughput measurements using synthetic job streams.
AFIPS Conf. Proc. (FJCC) 39:51-56, Nov., 1971.

[83] William A. Wulf, Roy Levin, and Samuel P. Harbison.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.

[84] W. Wulf, *et al.*.
BLISS-11 Programmer's Manual.
Digital Equipment Corporation, 1970.